NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ADAPTIVE METHOD FOR SCHEDULING THE SEQUENCE AND ROUTE OF BUILDER TRIALS FOR A NEW SHIP

by

Ahmed Raza Tahir

September 2015

Thesis Advisor:	Susan M. Sanchez
Second Reader:	William Solitario

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE AN ADAPTIVE METHOD FOR SCHEDULING THE SEQUENCE AND ROUTE OF BUILDER TRIALS FOR A NEW SHIP			**5. FUNDING NUMBERS**
6. AUTHOR(S) Tahir, Ahmed Raza			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Before a newly built ship is brought into service, it has to undergo various trials as part of its delivery. The builder shipyard aims at completing the maximum number of trials in the minimum possible time. Many trials have one or more prerequisites and every trial may have certain environmental requirements for its conduct. At sea, the success of any specific trial cannot be guaranteed. A dynamic tool is needed to help decision makers rapidly construct alternative trial sequences after the failure of any trial, and aid them in deciding whether the trials can be continued, or the ship has to return to the harbor for repairs. This thesis develops such an adaptive tool, which generates an optimal sequence and feasible route for conduct of a given set of trials, minimizing the total time required in the absence of failures or adverse environmental conditions. The tool allows the user to generate alternate sequences of trials if an early trial fails. Simulation of the conduct of trials, under varying environmental conditions, reveals that the number of retries is the most important factor affecting the outcomes. It also identifies bottlenecks in the network, providing insight about onboard spare supportability for important systems.

14. SUBJECT TERMS builder trials, optimal sequence and route, robust solution, layered network, shortest path, design of experiments, simulation			**15. NUMBER OF PAGES** 117
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ADAPTIVE METHOD FOR SCHEDULING THE SEQUENCE AND ROUTE OF BUILDER TRIALS FOR A NEW SHIP

Ahmed Raza Tahir
Lieutenant Commander, Pakistan Navy
B.E., Pakistan Navy Engineering College, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Approved by: Susan M. Sanchez
Thesis Advisor

William Solitario
Second Reader

Patricia A. Jacobs
Chair, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Before a newly built ship is brought into service, it has to undergo various trials as part of its delivery. The builder shipyard aims at completing the maximum number of trials in the minimum possible time. Many trials have one or more prerequisites and every trial may have certain environmental requirements for its conduct. At sea, the success of any specific trial cannot be guaranteed. A dynamic tool is needed to help decision makers rapidly construct alternative trial sequences after the failure of any trial, and aid them in deciding whether the trials can be continued, or the ship has to return to the harbor for repairs. This thesis develops such an adaptive tool, which generates an optimal sequence and feasible route for conduct of a given set of trials, minimizing the total time required in the absence of failures or adverse environmental conditions. The tool allows the user to generate alternate sequences of trials if an early trial fails. Simulation of the conduct of trials, under varying environmental conditions, reveals that the number of retries is the most important factor affecting the outcomes. It also identifies bottlenecks in the network, providing insight about onboard spare supportability for important systems.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. BACKGROUND AND MOTIVATION....2
B. THESIS OBJECTIVES....3
C. THESIS ORGANIZATION....3

II. LITERATURE REVIEW....5

III. METHODOLOGY....7
A. PLANNING PHASE....8
1. Generation of Sequence....8
2. Creation of Layered Network....8
3. Finding the Route to Complete All Trials....12
B. SIMULATION....13
C. FACTORS AND RANGES....14
D. ASSUMPTIONS....15
E. LIMITATIONS....16
F. DESIGN OF EXPERIMENTS....16

IV. RESULTS AND ANALYSIS....17
A. GENERAL RESULTS....17
1. Input Data File....17
2. Sequence of Trials....19
3. Generating Multiple Feasible Trial Sequences....19
4. Generating the Path....19
5. Out of Bound Areas....21
B. SIMULATION AND DESIGN OF EXPERIMENTS....22
C. MEASURES OF EFFECTIVENESS....25
D. SIMULATION RESULTS....26
1. Regression Model....30
2. Identification of Important Systems....31
E. SIMULATION RE-RUNS....32
1. Simulation Re-run Results....33
2. Regression Model for Simulation Re-run....35
F. COMPARISON OF RESULTS....36
G. DISCUSSION....39
H. MODEL ENHANCEMENTS....40

V. CONCLUSIONS AND RECOMMENDATIONS.....41

APPENDIX A. CONTENTS OF INPUT DATA FILE 43

APPENDIX B. PYTHON FUNCTION TO GENERATE THE SEQUENCE OF TRIALS 49

APPENDIX C. PYTHON CODE TO GENERATE MULTIPLE FEASIBLE SEQUENCES 51

APPENDIX D. PYTHON CODE TO GENERATE THE PATH FOR TRIAL CONDUCT 53

APPENDIX E. PYTHON CODE TO RUN SIMULATIONS.....61

APPENDIX F. PYTHON CODE TO CALCULATE BETWEEN-NESS AND CLOSENESS CENTRALITY ALONG WITH RESULTS 75

APPENDIX G. PYTHON CODE FOR SIMULATION RE-RUNS.....77

LIST OF REFERENCES 91

INITIAL DISTRIBUTION LIST 93

LIST OF FIGURES

Figure 1. General Flow of Events in the Model. 7

Figure 2. Major Steps in Planning Phase. 8

Figure 3. Flow Chart Showing the Creation of the Layered Network. 9

Figure 4. General Trials Area. 9

Figure 5. Trials Area Superimposed with Grid. 10

Figure 6. Portion of `depths.csv` File. 11

Figure 7. Illustration of Nodes and Edges in a Single Layer. 12

Figure 8. Illustration of the Layered Network. 12

Figure 9. Flowchart Showing the Broad Logic Scheme of Simulation Model. 13

Figure 10. Route Plot for 10 Trials. 21

Figure 11. Route Plot for 10 Trials with Interdiction Area. 22

Figure 12. Scatterplot Matrix of Input Factors. 24

Figure 13. Distribution and Summary Statistics of Input Factors. 25

Figure 14. Correlation Matrix for Input Factors. 25

Figure 15. Histograms and Summary Statistics of Mean and Standard Deviation of Number of Trials Completed. 26

Figure 16. Partition tree for Mean Number of Trials Completed as Response Variable. 28

Figure 17. Total Time vs. Number of Trials Completed Overlaid by Number of Retries. 29

Figure 18. Regression Summary for Mean Number of Trials Completed. 30

Figure 19. Closeness Centrality Plot. 31

Figure 20. Between-ness Centrality Plot. 32

Figure 21. Histograms and Summary Statistics of Mean and Standard Deviation of Number of Trials Completed. 33

Figure 22. Partition Tree with Mean Number of Trials Completed as Response. 34

Figure 23. Results of Regression Model. 35

Figure 24. Regression Model Results for Simulation Re-run. 37

Figure 25. Total Time vs. Number of Trials Completed, Overlaid by Number of Retries (k) when all Systems Given Same Number of Retries. Results of Subset Where Number of Trials Completed > 70. Simulation Runs with Lower Value of k not Seen Very Often. 38

Figure 26. Total Time vs. Number of Trials Completed, Overlaid by Number of Retries (*k*), when One Extra Retry is Given to the Ten Most Important Systems. Results of Subset Where Number of Trials Completed > 70. Simulation Runs with Lower Value of *k* Completing More Trials.....39

LIST OF TABLES

Table 1. Factors and Ranges Used in the Simulation Experiment. 15

Table 2. Portion of `Data.csv` File. 18

Table 3. Sequence to Conduct all Trials. 19

Table 4. Route to Conduct All Trials. 20

Table 5. Design Points for the Simulation, where the Factors are: *n* (Total Number of Acceptable Failures), *k* (Number of Retries per Trial), Wind State, Sea State, and Speed of the Ship. 23

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BRAC	Base Realignment and Closure
CSV	Comma Separated Value
DOE	Design of Experiments
DoN	Department of the Navy
GL	Germanischer Lloyd
GUI	Graphical User Interface
HLA	High Level Architecture
KS&EW	Karachi Shipyard & Engineering Works
MOE	Measure of Effectiveness
NOLH	Nearly Orthogonal Latin Hypercube
NSRP	National Shipbuilding Research Program
RTI	Run Time Infrastructure
SHIP	Ship and Installation Program
SOLAS	Safety of Life at Sea
US	United States

THIS PAGE INTENTIONALLY LEFT BLANK

THESIS DISCLAIMER

The reader is cautioned that the computer programs presented in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logical errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Before a newly built ship is brought into service, it has to undergo various trials as part of its delivery from the builder shipyard to the owner. Many types of equipment and performance are tested during the delivery trials. These trials are generally conducted in two phases, i.e., harbor trials and sea trials. This research focuses only on the sea trial phase. These trials are conducted to verify the design parameters of the ship, the operability of main and auxiliary machinery, and the weapons and sensors for warships. The builder shipyard aims at minimizing the total time to conduct the sea trials, as it involves expenses such as fuel, crew, trial equipment, trial team, and food. Many trials have one or more prerequisites. For example, a prerequisite for conducting a speed test might be successful completion of an engine test. Moreover, every trial may have certain environmental requirements for its conduct, such as a minimum water depth, sea state limitations, or maximum wind force.

This gives rise to a complex network problem, considering that there are multiple sequences in which these trials can be conducted and each trial requires the ship to be routed to an appropriate position at sea. While at sea, the success of any specific trial cannot be guaranteed, and therefore a dynamic tool is needed to help decision makers rapidly construct alternate trial sequences after the failure of any trial, and aid them in deciding whether the trials can be continued or the ship has to return to harbor for repairs.

This thesis develops such an adaptive tool, which generates an optimal sequence and feasible route for conduct of a given set of trials, minimizing the total time required in the absence of failures or adverse environmental conditions. The tool allows the user to generate alternate sequences of trials if an early trial either cannot be attempted, or is attempted but unsuccessful. After generating the sequence of trials, important systems in the network are identified based on their measures of closeness centrality and betweenness centrality. These measures give the importance factor for nodes in the network based on number of edges and number of shortest paths a node is on. Figure 1 shows the results plotted for a small-scale model run using only ten trials.

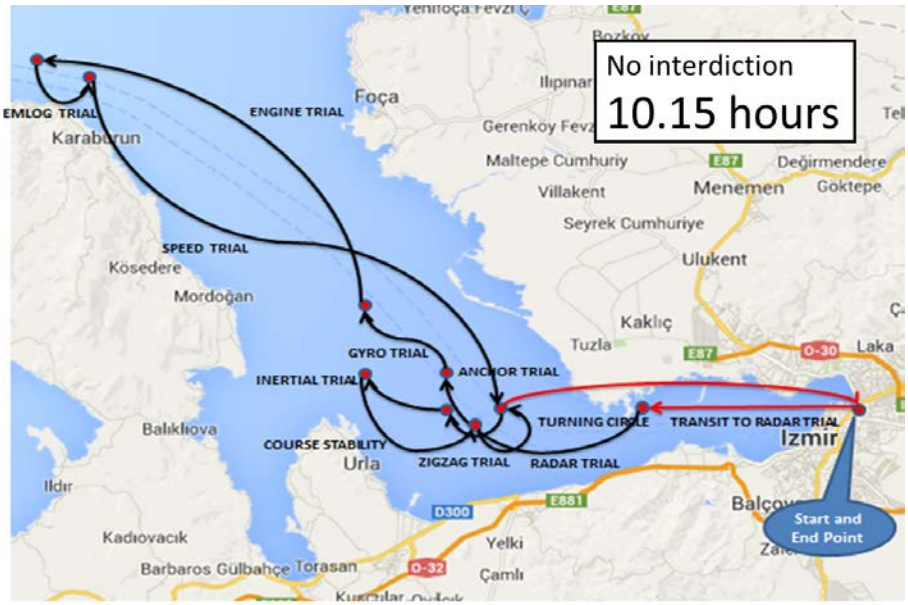


Figure 1. Small-scale model run using ten trials.

After successful development of this scheduling and routing tool, the sea trials phase is simulated using designed experiments to study the effects of environmental conditions, namely sea state and wind force, as well as the total number of acceptable failures (n), number of retries for each trial in case of its failure (k), speed of transit, and probability of success for each trial.

The primary measure of effectiveness (MOE) is the number of trials completed before returning to port. Simulation of the sea trial process under varying environmental conditions, specified by a designed experiment, provides the key insight that number of retries for each trial (k) is the most significant input factor for our MOE of number of trials completed. The results may be translated as: the higher the number of retries given to each system, the higher will be the number of trials completed. However, this is not a feasible option in actual practice, as ships will end up spending a very long time out at sea if too many retries are allowed. Therefore, a second experiment re-runs the simulation by giving one extra retry to the ten most important systems. This extra retry can be translated into the spares availability onboard for important systems, and gives the important systems an extra retry for conduct. Descriptive statistics, partition trees, and regression models from the simulation re-run show that allowing one extra retry for important systems leads to a substantial increase in the number of trials completed.

Following are the summarized results of this research:

- The tool may be used in the initial planning phase to find a sequence to conduct trials, taking into consideration all the prerequisites to complete those trials.

- The tool may be used in the initial planning phase to find a route for this sequence, minimizing the time to complete all trials in the absence of failed attempts or adverse environmental conditions.

- The simulation results reveal the distribution of outcomes possible if failed attempts are possible or adverse environmental conditions are present. This information may also be useful in the initial planning phase.

- The number of retries per trial (k) is the most important determinant of the number of trials completed before the ship returns to port.

- The tool may be used to identify important systems for spare supportability. This will in essence give the important systems one extra retry (k) in case of a failed attempt.

- In case of failure of a system at sea, the tool may be used to generate an alternate sequence for conduct of trials for remaining systems.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First of all, I would like to express my deepest appreciation to Professor Susan Sanchez for her patience, simulation and data analysis insights, valuable suggestions, and being a great mentor throughout my research. Without her guidance and support it would be hard to complete this thesis. I would also like to thank my second reader, Professor William Solitario, for his support and providing me operational insights, especially during the initial development phase. I would also like to thank Professor Ned Dimitrov, who provided me with valuable ideas when I first introduced the problem to him. I would like to thank Steve Upton for his support in running large experiments on the SEED lab cluster computer.

I would like to thank my wife, Asma, for her endless patience and encouragement during this time. She never complained about taking care of our lovely boy, Muhammad, alone, and I always found a smiling face from her when I came home. Without her precious love, this time would be hard for me.

Finally, to my parents who always prayed for my success and have made me the person I am today.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The shipbuilding industry in Pakistan has been limited to smaller ships and tugboats in the past. The Karachi Shipyard is the major shipyard in the country, and most of the shipbuilding work being done is for the Pakistan Navy and the Karachi Port Authority. Vessels built by the Karachi Shipyard include smaller missile boats, ocean tugs, corvettes, submarines, and in the recent past frigates. The Karachi Shipyard is also involved with repair, rebuilding, and overhauling of naval and commercial vessels (http://www.karachishipyard.com.pk/).

Before bringing a newly built ship into service, it has to undergo various trials as part of its delivery from the builder shipyard to the owner. The conduct of builder trails is a complex process. There are hundreds of systems on a ship and each system has its own prerequisites that include, but are not limited to, completion of other trials before attempting that trial, sea state and wind requirements, and specific water depth requirements. While at sea, if a system fails during the conduct of its trial, the decision makers are faced with the problem of deciding whether to stay out at sea and continue with remaining trials, or proceed back to port to fix the failed system and return for conduct of trials. This decision often becomes the source of long debates as a number of stakeholders are involved. The builder shipyard aims at completing the maximum possible trials at sea, while the ship crew does not want to miss any prerequisites and therefore tends to stick with the original plan. There is no adaptive tool present with the decision makers at sea which can provide them insight into the future possibilities and help them make the decision whether to stay out at sea or return to port.

This thesis deals with development of an adaptive tool to generate an optimal sequence, and possible feasible route to be taken, for conduct of a given set of trials minimizing total time for conduct of trials. After successful development of this scheduling and routing tool, the process of conduct of trials is simulated in a structured way, applying a design of experiments approach that varies environmental conditions to develop a robust optimization solution for the sequence of conduct of these trials and the route to be taken.

A. BACKGROUND AND MOTIVATION

Before bringing a newly built ship into service, it has to undergo various trials as part of its delivery from the builder shipyard to the owner. Many types of equipment and performance are tested during these delivery trials (Haakenstad, 2012). These trials are generally conducted in two phases, i.e., harbor trials and sea trials. The sea trials are conducted to ensure that the system is in compliance with the standards, and meets the claims made by the manufacturer regarding system specifications. These trials include the validation of ship design based on parameters such as stability and seakeeping ability, maximum speed, and fuel consumption (Hart, 2000). Warships have more systems installed on them compared to the merchant vessels. Additional systems may include but are not limited to weapons, sensors, auxiliary systems, and auxiliary machinery to support these weapons and sensors. This thesis research focuses on the sea trials phase only.

For the conduct of sea trials, the builder shipyard aims at minimizing the total time to conduct these trials, as it involves expenses such as fuel, crew, trial equipment, trial team, food, etc. In certain cases, the trials have prerequisites to be completed. For example, a prerequisite can be completion of engine test before conduct of speed test. In many cases there are multiple prerequisites to be completed. Moreover, every trial may have certain environmental requirements for its conduct, for example minimum water depth requirement, sea state and wind force, etc.

This gives rise to a complex network problem, considering the fact that there are multiple sequences in which these trials can be conducted and each trial requires the ship to be in a certain position at sea. While at sea, the success of any specific trial cannot be guaranteed. In the case of an unsuccessful trial for any system at sea, the decision makers are faced with the dilemma of future plans. Options are very limited while at sea; they may include fixing and retrying the trial, leaving the failed trial and proceeding with the remaining trials, or in extreme cases returning to port to fix the failed system. At present, there is no tool available to decision makers while they are out at sea to reschedule the possible sequence after removing the failed trial from the list. Decision makers also need an updated feasible route for conduct of remaining trials. This thesis takes the motivation

from this problem and provides the decision makers with a dynamic tool that can generate alternate optimal sequences and routes for conduct of trials in case of failure of a given system. The tool also gives the decision makers visibility on remaining trials. This information aids the decision maker with their decision to stay out at sea or return to port.

B. THESIS OBJECTIVES

This research focuses on effective and adaptive scheduling of the sequence of builder trials for a newly built ship, along with the route to be taken. The result is aimed at reducing the total time taken to conduct the trials, thereby reducing the overall cost of the trials. Following are the broad research questions:

- What are possible sequences to conduct builder trials for a newly built ship with the objective of minimizing the time at sea subject to the prerequisites and specific trial requirement for a given set of conditions?
- What are possible alternate sequences in case some trials are deleted from the list due to any reason such as system failure?
- What is an optimal route to conduct the generated sequence of trials? For a given mix of trials and sea conditions, how variable are the results if the initial sequence is used?
- What are the results of simulating the process of conduct of trials by applying design of experiments (DOE) to the model? Factors such as wind force, sea state, permissible number of failures and retries, and probability of success for each trial will be varied for DOE.

This research develops a dynamic tool which can be used for finding the sequence and route for conduct of builder trials for a ship. The study also identifies the important factors that have the greatest impact on the time taken to complete trials and the number of trials completed in a single trip to sea.

C. THESIS ORGANIZATION

The study uses a simulation model and state-of-the-art design of experiments to quantify the risk associated with conduct of builder trials. Chapter II is a literature review that identifies past research on this topic. Chapter III discusses the methodology. It examines the model structure, variables, constraints, limitations, and assumptions, and provides details about the design of experiments used to make multiple runs of the model.

Chapter IV deals with analysis of the results. Chapter V is conclusion. It also gives recommendations and associated future work with the research.

II. LITERATURE REVIEW

This chapter explores the past work that has been done with regards to the conduct of trials for newly built ships. Although there is no directly related work in the past, most of the works done are in areas of ship building, shipyard design to improve the shipbuilding process, and procedures and requirements for conduct of trials for newly built ships. A brief description of past works is shown in ensuing paragraphs.

Carter (2005) discusses the shipbuilding program history for the U.S. Navy, and analyzes case studies explaining the importance of system integration during the process of shipbuilding. She presents both cases with successful system integration, and not so successful as well. The research discusses the ideas and concepts to simplify the process of ship design and system integration. Successful system integration during the design phase may reduce the need for major design changes during the building process.

Colgary and Willet (2006) use an integer linear program to study the Navy's Configuration Analysis, which includes the surface and subsurface vessel stationing problem. Their Ship and Installation Program (SHIP) calculates the minimum cost ship stationing requirements, while maintaining the operational requirements constraints. SHIP has the ability to propose future force structure disposition as well. Their work might be combined in the future with this research for large-scale, multi-ship trial scheduling.

McLean and Shao (2001) discuss the objectives and requirements for a shipbuilding simulation. The research work mainly involves a generic simulation of the shipbuilding operation. This simulation model helps identify scheduling conflicts with regard to job completion, as well as various resource allocation problems that can arise during the shipbuilding operation. The results help identify the requirements of new technologies, especially with regards to scheduling and cost.

The National Shipbuilding Research Program (NSRP) (1999) examines the rules and regulations for commercial ships inspection, and identifies differences and similarities within existing requirements from several standards organizations. The main

results of the study are a sample trial database for a commercial ship, plans for conduct of the trials, and comparison matrices for highlighting the differences and similarities in the existing regulations. One of these standards organizations, Germanischer Lloyd (2012), provides general guidelines for sea trials before commissioning a vessel. The guidelines in these rules provide all the stakeholders with an overview of the process of sea trials which is required to fulfill the requirements of Germanischer Lloyd (GL) and the International Convention for the Safety of Life at Sea (SOLAS). The data organization in this thesis is broadly based on the GL rules and NSRP procedures.

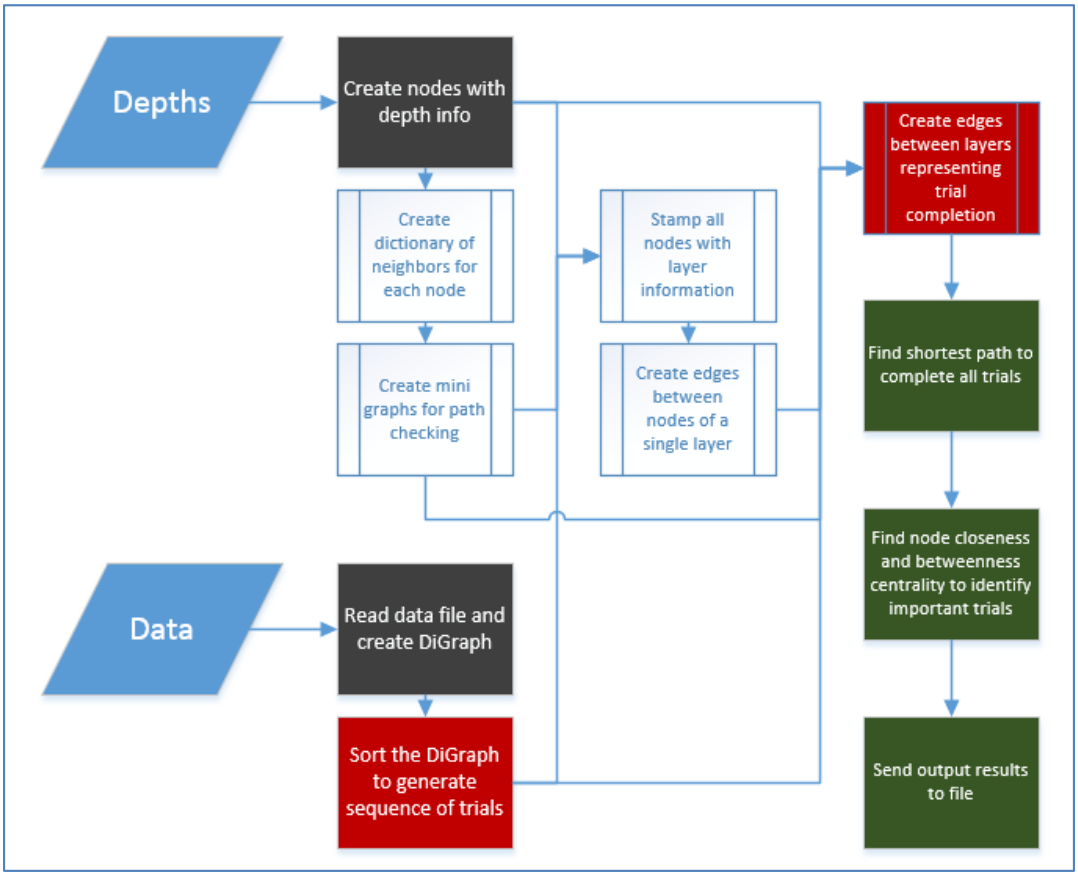
Haakenstad (2012) provides a general discussion on the process of sea trials of ships. The research mainly deals with the conduct of speed trials of the ship and the process of applying corrections to the speed trials. She discusses in detail how the corrections are applied to speed trials and also discusses various methods to apply these corrections. She also compares the results from corrections applied by shipyards with results from corrections applied based on standards, and discusses the impact of these differences on final results. This underscores the importance of considering environmental and atmospheric conditions on sea trials. This thesis analyzes the general impact of these and other factors on trial success, rather than detailed effects on individual trials.

There has been a tremendous amount of work done in the past with regard to the shipbuilding process; however, there is no tool available which deals with the complete process of sea trials. The previous work either deals with individual trials (Haakenstad, 2012) or focuses on the process of shipbuilding rather than testing (Carter, 2005; McLean and Shao, 2001). The research in this thesis is aimed at consolidating the process of scheduling the process of sea trials and additionally providing the route for conduct of those sea trials. The adaptive nature of the tool allows the user to regenerate the sequences and routes in case of system failures at sea.

III. METHODOLOGY

The simulation developed for this thesis models the process of planning and conduct of builder trials for a newly built ship. The planning phase consists of generating a sequence to conduct the trials and finding the route to conduct those trials, while keeping track of the sea depth required for each trial and other prerequisites. The conduct phase uses the information from the planning phase and simulates the conduct of trials under varying operational factors, including probability of success for each trial. Both the planning and conduct phases are developed using the Python programming language (Python website). Figure 1 shows the sequence of events in the model.

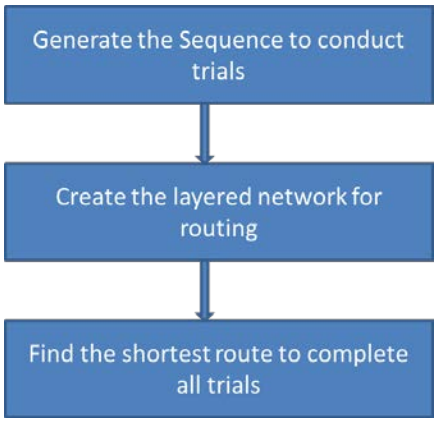
Figure 1. General Flow of Events in the Model.



A. PLANNING PHASE

The major steps in the planning phase are shown in the flowchart in Figure 2, and then described in more detail.

Figure 2. Major Steps in Planning Phase.



1. Generation of Sequence

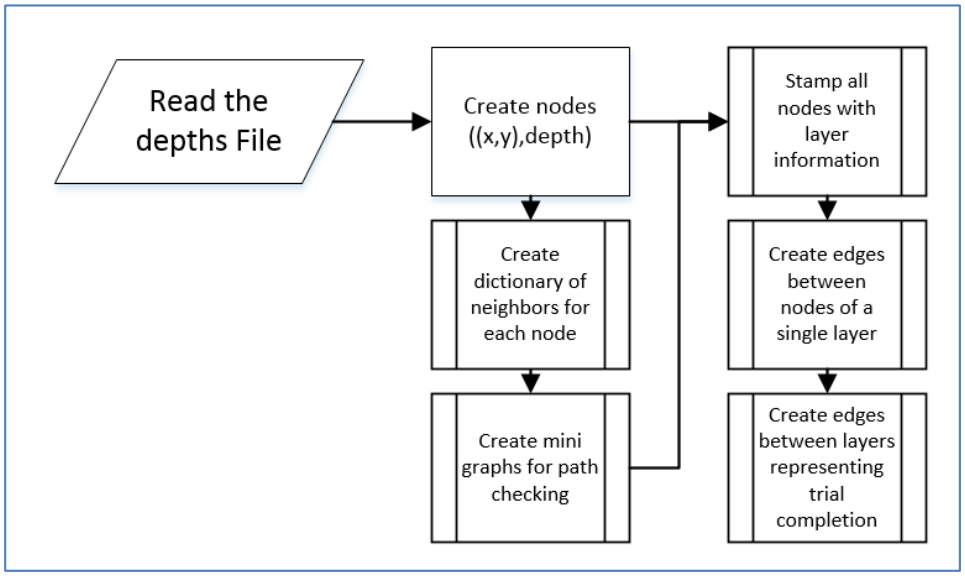
This portion of the program takes the `Data.csv` file as input. The `Data.csv` file contains the trial information, which includes the trial name, time to complete, prerequisites for the trial, average speed during conduct, minimum sea depth required, sea state and wind force requirements, and mean and standard deviation of the distribution of probability of success (discussed further in Section B). Complete contents of `Data.csv` are shown in Appendix A.

After reading the data file, the Python program (modified Dimitrov, 2014) creates a directed graph that takes into account all the prerequisites for all trials. A topological sort of the graph generates the sequence to conduct the trials. See Appendix B for details of the function to generate the sequence.

2. Creation of Layered Network

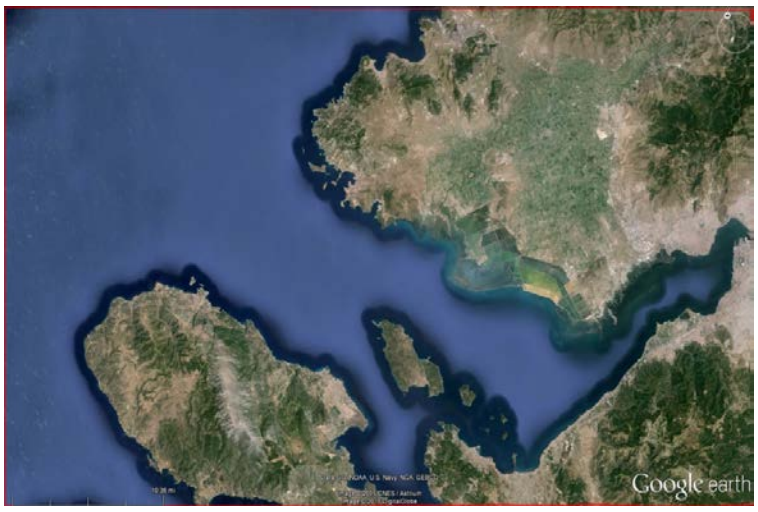
In order to find the shortest path to conduct all trials, a layered network is created. The flow chart in Figure 3 shows the sequence of events for generation of the layered network.

Figure 3. Flow Chart Showing the Creation of the Layered Network.



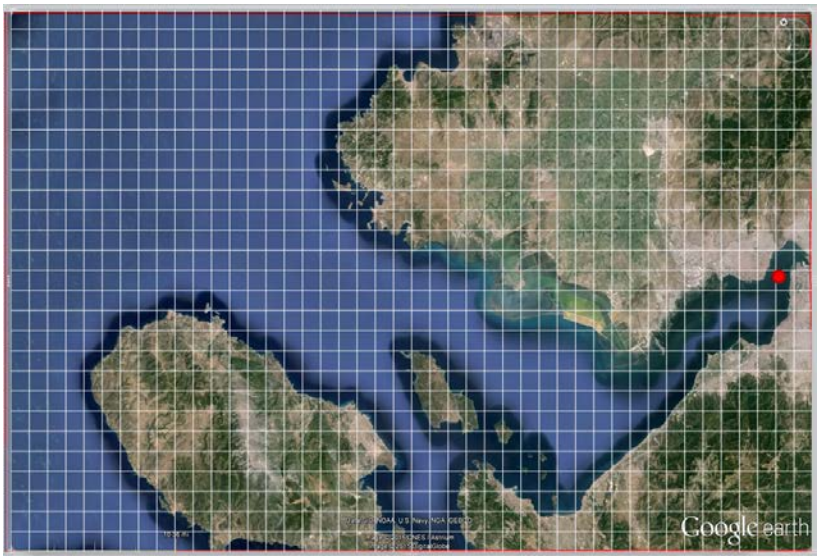
The input file is named `depths.csv` for this phase. The `depths.csv` file contains depth information for the trial area, which has been divided into 1x1 Nautical Mile grid boxes. Depth is noted at every grid point and is recorded in the `depths.csv` file. Figure 4 shows the general trials area, and Figure 5 shows trial area superimposed with the grid.

Figure 4. General Trials Area.



From Google Earth Pro (2015, August), Izmir, Turkey [Terrain Map]. Retrieved from Google Earth Pro Version 7.1.2.2041 on August 15, 2015.

Figure 5. Trials Area Superimposed with Grid.



After Google Earth Pro (2015, August), Izmir, Turkey [Terrain Map]. Retrieved from Google Earth Pro Version 7.1.2.2041 on August 15, 2015.

The trial area map has been chosen from Google earth. It shows the Izmir harbor in Turkey, however, depths recorded in the `depths.csv` file are not actual charted depths but approximations to nearest charted depths. The red dot represents the starting and ending location for the trials. The grid is represented as Cartesian coordinate (x,y) pairs, and numbering starts from top left as position (1,1). Figure 6 shows a portion of the `depths.csv` file.

Figure 6. Portion of `depths.csv` File.

75	0	0	0	0	0	0	0	0	0	0
75	0	0	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0	0	0
66	0	0	0	0	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0	0	0
38	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
47	20	0	0	0	0	0	0	0	0	0
56	47	0	0	0	0	0	0	0	0	0
63	55	37	17	0	0	0	0	0	0	0
67	63	42	31	15	0	0	0	0	0	0
60	67	53	21	7	5	0	0	0	0	0
50	65	55	35	3	7	0	0	0	0	0
0	55	62	62	16	15	0	0	0	0	0
0	40	57	65	30	18	4	0	0	0	0
0	27	50	73	55	25	7	4	0	0	0
0	0	47	67	62	52	25	7	4	0	0
0	0	30	50	67	57	45	18	7	4	0
0	0	0	52	60	65	52	46	16	10	0
0	0	0	35	62	67	54	51	52	11	6
0	0	0	0	47	53	53	50	45	45	40
0	0	0	0	35	48	45	45	35	30	27
0	0	0	0	20	30	30	27	0	0	0

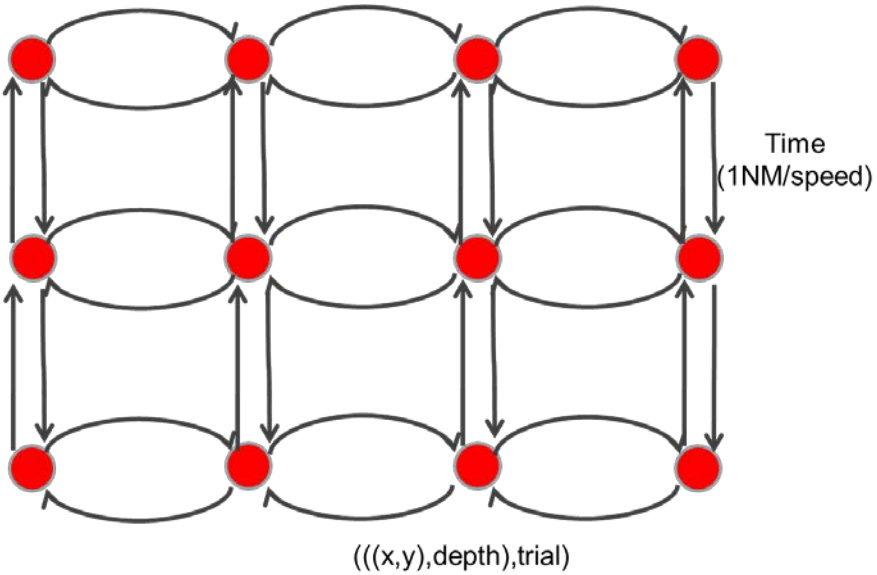
Every number is noted as a Cartesian coordinate in the grid with top left of the area as position (1,1). All depths noted as ‘0’ represent land or areas not safe for navigation. The user needs the dimensions of the grid (number of rows and number of columns) as input parameters while reading the depths file into python.

The program reads in the depths file and, based on the sequence created in planning phase, generates a layered network. Edges in a single layer represent the transit between grid locations. The edge cost between nodes of a single layer is calculated as follows:

$$Edge\_Cost = \frac{1.0NM}{Speed}$$

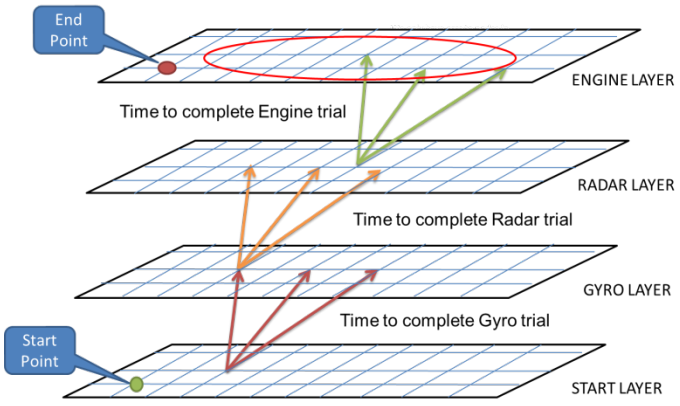
where ‘Speed’ represents the transit speed of the ship in knots while transiting to adjacent node in the grid. Figure 7 illustrates nodes and edges within a single layer.

Figure 7. Illustration of Nodes and Edges in a Single Layer.



Each layer in the network represents a single trial over the entire trial area. Edges between layers in the graph represent trial completions. The edge cost between layers is the time taken to complete the trial. Figure 8 shows an illustration of the layered network.

Figure 8. Illustration of the Layered Network.



3. Finding the Route to Complete All Trials

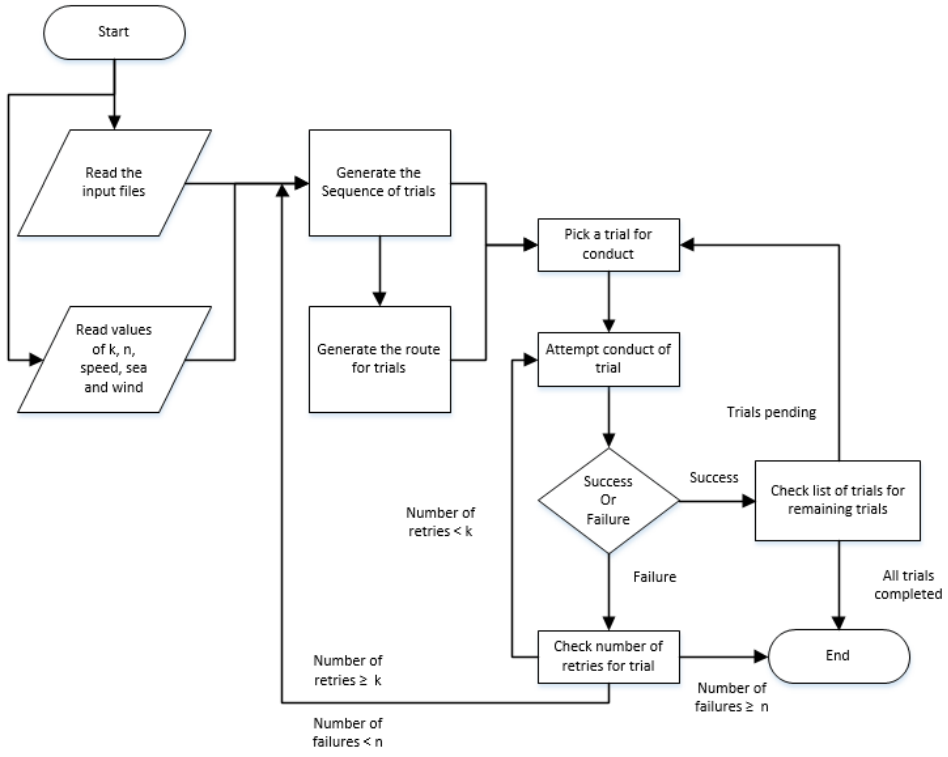
After successful generation of the layered network, the model finds the shortest path for completion of all trials. The model also gives the minimum time taken to

complete the trials. This is done by using the built-in shortest_path function in the networkx module of Python. The result is a list of nodes that represents completions of trials at specific locations at sea. The time to complete all trials provided by the model assumes that all trials are successful on the first attempt, so it serves as a lower bound on the time it will take to complete all trials when failed attempts are possible.

B. SIMULATION

The simulation phase includes generating the sequence of trials, creating the layered network, finding the route to conduct the trials, and finally simulating the process of conduct of trials under varying input conditions. The flowchart in Figure 8 shows the sequence of events for the model.

Figure 9. Flowchart Showing the Broad Logic Scheme of Simulation Model.



The first step in the simulation reads the user inputs for values of *k* (number of retries allowed per trial), *n* (Total number of trial failures acceptable), sea state (between

1 and 5), wind force (between 1 and 6), and speed of transit. After these values have been recorded, the data.csv and depths.csv files are read in, and the sequence of trials and route for trials are generated. Thereafter, the conduct of each trial in sequence is simulated. The probability of success for a specific trial is drawn from an approximately normal distributed with a user-specified mean and standard deviation. (The approximation arises because any generated values below zero are set to zero, and any generated values above one are set to one.) The provision for allowing unique probability distributions for each trial is kept because in the real world, each trial has a separate distribution for probability of success. Sometimes these can be created from previous trial data. Therefore, probability of success can be changed for future research where no or very slight modifications in the program will enable the user to read in specified probabilities of success for each trial. In this thesis, for a single run of simulation, a separate probability of success for each trial is generated from the specified distribution. The success or failure of a particular attempt is adjudicated based on the drawn random number compared with the probability of success for that trial. If a trial attempt fails and the current number of retries for that trial is less than *k*, the trial conduct is retried. If number of retries has reached the value of *k*, then the trial is removed from the network, a new sequence of trials is generated that keeps track of all completed trials along with the ones that cannot be completed because of removal of previously failed trials. Once the graph is updated and a new sequence and route has been generated, the process of conduct of trials is repeated as described above. Also, when a trial fails and the total number of failed trials has reached the value of *n* (Total number of failures acceptable), the simulation stops and the ship returns to port.

C. FACTORS AND RANGES

The simulation has five input variables, or factors, that are explored. The probability of success for each trial for each simulation run is different as it is a random draw from an approximate Normal distribution with fixed mean and standard deviation. The factors are described in Table 1. The high and low levels of the factors are specified within reasonable acceptable ranges by subject matter experts. Values contained in the

`Data.csv` file are based on the author's previous experience as Navigation Officer and involvement in builder trials of newly built ships.

Table 1. Factors and Ranges Used in the Simulation Experiment.

Input Variable	Description	Min Value	Max Value
n	Total number of failures acceptable before returning to port (1 failure = failure of an entire trial after allowed retries)	5	15
k	Number of retries per trial before moving to next	2	5
sea	Prevailing sea state	1	5
wind	Prevailing wind force (Beaufort Scale)	1	6
speed	Speed of transit	10 kn	25 kn

D. ASSUMPTIONS

The main assumptions in the model are made in order to scope the problem in a reasonable manner. The assumptions are listed below:

- Traffic does not hamper the conduct of trials.
- Trials are conducted in a limited sea area.
- Probability of success for all trials is ~Normally distributed with fixed μ and σ, where $0< \mu <1$, $\sigma / \mu << 1$, and the probability is truncated outside the interval [0,1].
- Ship only makes one trip to sea to attempt conduct of all trials. Simulation of conduct does not necessarily conduct all trials and may result in ship returning to port after acceptable failure cap (*n*) has reached.
- Every trial can be retried *k* times before moving to the next trial.
- The decision to return to port is based on value of *n*, which is the fixed maximum number of allowed failures.
- Unsuitable weather conditions reduce the probability of success for each trial by half.

- Areas to be avoided are represented as fixed boxes, and remain out of bounds throughout the process of conduct of trials.

E. LIMITATIONS

The development of this model is a first effort at the process of generating the sequence and route for builder trials for a newly built ship. The model is an endeavor to closely match the real-life process of conduct of builder trials; however, not all aspects are represented in the model. The limitations in the model are numbered and listed below:

- The simulation does not run through the completion of all trials. If the number of failed trials reaches the allocated threshold, the ship returns to port and the simulation stops. In reality, the ship will fix the failed systems after returning to port and proceed to sea again, repeating this process until all trials are complete.

- In reality, all trials have different distributions for probability of success. This aspect has been simplified in the model by assuming identical distribution with fixed values of mean and standard deviation.

- The model does not allow for simultaneous conduct of multiple trials. In reality, certain systems have no prerequisites, and can be tested simultaneously.

F. DESIGN OF EXPERIMENTS

The design of experiments uses the Nearly Orthogonal Latin Hypercube (NOLH) design spreadsheet (Sanchez, S. M. 2011, NOLHdesigns_V6 spreadsheet) based on the NOLH designs of Cioppa and Lucas (2007). The high and low values of for the five factors from Table 1 are entered into the NOLHdesigns_V6.xlsx design spreadsheet, and design points are generated by stacking and rotating the 17-design point design three times. This yields a total of 49 distinct design points, because all three stacks have the same center point. After copying the design points into a comma separated value (CSV) file, simulation runs are made by replicating each design point ten times. This stacked NOLH design limits the maximum amount of pairwise correlation, while achieving good space filling of the regions of interest, for the set of input factors.

IV. RESULTS AND ANALYSIS

The intent of this chapter is to identify and examine results obtained. First, we generate the sequence to conduct trials followed by generating a feasible route. We also identify multiple feasible sequences to conduct all trials. Thereafter, the process of conduct of trials is simulated and various measures of effectiveness are measured for each run for analysis. After analysis of the results obtained from simulations, we identify the ten most important trials in the network and give those trials one extra retry for simulation of conduct of trials. Finally, we re-run the simulation based on identified important systems and evaluate the performance of the process of conduct of trials. All graphs and models are generated using JMP Version 12.

A. GENERAL RESULTS

1. Input Data File

A portion of the contents of the input data file used is shown in Table 2. Complete contents of input data file are attached as Appendix A. The data contained in this file do not represent actual information, but are notional data for classification reasons. The prerequisites for trials are defined based on the author's previous experience with the process of conduct of builder trials. The 'Trial' column gives the name of the trial, and the 'ID' column shows the ID (three letter acronym) assigned to that trial. 'Time' represents the time required in minutes to complete that trial, and 'Depth' shows the minimum depth required for conduct of that trial. 'Prerequisites' shows the prerequisite trials that must be completed before conduct of that particular trial. 'Speed' gives the average speed of the ship in knots during the conduct of the trial. The 'Mean' and 'SD' columns give the mean and standard deviation of the probability distribution used to generate the probability of success for each trial, as discussed earlier. The 'Wind' and 'Sea' columns gives the maximum desirable wind force and sea state for conduct of that trial. In a case where the prevailing wind and sea state at the time of conduct are more than these limits, the probability of success for that trial is reduced by half.

Table 2. Portion of `Data.csv` File.

Trial	ID	Time	Depth	Prerequisites	Speed	Pdist Mean	Pdist SD	Wind	Sea
Engine	ENG	105	50	GY2;RDR; PGS;MSB; EGS;ESB; TLI;CO2; FF1;SWS; EMS;ECS; EEX	20	0.65	0.13	2	2
Fin Stabilizer	FIN	60	50	ENG	20	0.65	0.13	3	4
Compartment Noise	CNS	120	100	ENG;EEX; AC1;VNT; SWT;CHW; RFG;SWS; CAS;PGS	15	0.65	0.13	1	1
Domestic Appliances	DAP	60	20	GWD	5	0.65	0.13	5	4
Doors Windows Hatches	DWH	45	20		10	0.65	0.13	5	4
Crane	CRN	45	20	PGS;MSB	5	0.65	0.13	2	2
ICCP Equipment	ICP	30	20	PGS;MSB	10	0.65	0.13	4	3
Marine Growth Prevention System	MGP	30	20	PGS;MSB	10	0.65	0.13	4	3
Liquid Tank Level Indications	TLI	30	20		10	0.65	0.13	3	2
Alarm System	ALM	60	20	PGS;MSB	10	0.65	0.13	4	3
Engine Exhaust Flaps	EEX	45	50	EMS;ECS; PGS;MSB	20	0.65	0.13	4	2
FF System	FF1	90	20	CO2;SPR; SWS;FWS; PWS	10	0.65	0.13	4	3
Sprinkling System	SPR	60	50		10	0.65	0.13	4	3
CO2 Fire Extinguishing System	CO2	45	50		10	0.65	0.13	4	3

2. Sequence of Trials

The Python code shown in Appendix B can be used for generating the sequence in which to conduct the trials. The output from that code, shown in Table 3, is a sequence for conducting all trials that takes into consideration all the prerequisites.

Table 3. Sequence to Conduct all Trials.

```
['Start', 'DWH', 'CO2', 'PGS', 'GY1', 'GY2', 'RDR', 'TLI',
'HFE', 'MSB', 'CAS', 'HYD', 'TOW', 'LOS', 'DOS', 'SWS',
'SWC', 'SMA', 'NSL', 'DGS', 'CRN', 'CDW', 'ECS', 'MGC',
'ESS', 'ESD', 'ICS', 'ICP', 'MGP', 'ANC', 'UHF', 'VHF',
'BLR', 'SBT', 'MHF', 'FTS', 'LSE', 'EMS', 'EEX', 'GPS',
'MTR', 'NTX', 'STS', 'MRE', 'CHW', 'ALM', 'CDE', 'CMS',
'SAT', 'RAS', 'VNT', 'SPR', 'BWS', 'SWT', 'FWS', 'GWD',
'DAP', 'AC1', 'RFG', 'PWS', 'FF1', 'CCT', 'EGS', 'ESB',
'PDS', 'BCD', 'SPT', 'ELL', 'ENG', 'CNS', 'SHV', 'EML',
'SPD', 'ZZ1', 'UWN', 'CS1', 'IN1', 'AIS', 'INC', 'ECD',
'CBT', 'FCS', 'CWS', 'GUN', 'OAV', 'CRA', 'TC1', 'FIN',
'AMV', 'End']
```

3. Generating Multiple Feasible Trial Sequences

As the sequence generated above utilizes a topological sort on a directed graph, there may be multiple possible sorts in the same graph. Each of the possible topological sorts in the graph represents a feasible sequence for conduct of these trials. The Python code of Appendix C can be used to enumerate all the possible combinations of sequences of trials in the graph. For this thesis, only the sequence shown in Table 3 is explored further.

4. Generating the Path

After generating the sequence to conduct the trials, the Python code shown in Appendix D was used for generating the path to conduct all the trials. Table 4 shows the path generated by the network. Each location is represented as (((x,y), Depth), Trial). In most cases, the ((x,y), Depth) represents the coordinates and depth where the trial

completes, and Trial provides the ID of the scheduled trial. The Trial ID repeated multiple times represents the transit from port to trial area ('Start' in these results) and transit back to port ('AMV' in these results).

Table 4. Route to Conduct All Trials.

Outbound Transit and Trials 1-23		Trials 24-55		Trials 55-87		Trial 88 and Return Transit	
Location	Trial	Location	Trial	Location	Trial	Location	Trial
((38, 38), 9)	Start*	((30, 5), 69)	ESS	((11, 11), 149)	DAP	((20, 9), 101)	AMV**
((38, 37), 10)	Start*	((29, 1), 119)	ESD	((6, 10), 141)	AC1	((21, 10), 97)	AMV**
((39, 36), 17)	Start*	((32, 3), 142)	ICS	((1, 10), 119)	RFG	((22, 11), 78)	AMV**
((38, 35), 6)	Start*	((35, 3), 129)	ICP	((9, 12), 68)	PWS	((23, 12), 72)	AMV**
((39, 34), 14)	Start*	((40, 3), 53)	MGP	((2, 4), 108)	FF1	((24, 13), 73)	AMV**
((38, 33), 3)	Start*	((40, 3), 53)	ANC	((2, 6), 144)	CCT	((25, 14), 78)	AMV**
((39, 32), 5)	Start*	((42, 3), 52)	UHF	((3, 1), 129)	EGS	((26, 15), 67)	AMV**
((39, 31), 7)	Start*	((42, 1), 93)	VHF	((4, 5), 149)	ESB	((27, 16), 62)	AMV**
((40, 30), 39)	Start*	((39, 1), 75)	BLR	((7, 13), 61)	PDS	((28, 17), 63)	AMV**
((40, 25), 50)	DWH	((41, 2), 83)	SBT	((12, 15), 64)	BCD	((29, 18), 63)	AMV**
((38, 22), 60	CO2	((42, 1), 93)	MHF	((15, 17), 79)	SPT	((30, 19), 63)	AMV**
((30, 19), 63)	PGS	((39, 3), 92)	FTS	((18, 17), 72)	ELL	((31, 20), 53)	AMV**
((31, 14), 63)	GY1	((42, 1), 93)	LSE	((8, 1), 109)	ENG	((32, 21), 35)	AMV**
((28, 10), 51)	GY2	((29, 3), 100)	EMS	((15, 11), 110)	CNS	((33, 22), 16)	AMV**
((26, 6), 77)	RDR	((26, 11), 65)	EEX	((29, 2), 110)	SHV	((34, 23), 18)	AMV**
((22, 6), 86),	TLI	((20, 12), 74)	GPS	((37, 2), 77)	EML	((35, 24), 7)	AMV**
((21, 5), 81)	HFE	((22, 17), 60)	MTR	((13, 16), 54)	SPD	((36, 25), 7)	AMV**
((11, 5), 105)	MSB	((17, 17), 79)	NTX	((13, 4), 150)	ZZ1	((37, 26), 7)	AMV**
((7, 10), 145)	CAS	((5, 13), 87)	STS	((4, 2), 134)	UWN	((38, 27), 10)	AMV**
((13, 7), 109)	HYD	((5, 9), 139)	MRE	((11, 8), 132)	CS1	((39, 28), 6)	AMV**
((12, 2), 122)	TOW	((1, 7), 142)	CHW	((11, 14), 93)	IN1	((40, 29), 34)	AMV**
((13, 5), 143)	LOS	((4, 11), 107)	ALM	((14, 16), 53)	AIS	((39, 30), 6)	AMV**
((18, 5), 135)	DOS	((4, 9), 110)	CDE	((12, 13), 75)	INC	((40, 31), 17)	AMV**
((10, 4), 122)	SWS	((3, 11), 123)	CMS	((5, 14), 68)	ECD	((39, 32), 5)	AMV**
((8, 1), 109)	SWC	((1, 11), 146)	SAT	((26, 3), 100)	CBT	((40, 33), 11)	AMV**
((6, 2), 147)	SMA	((18, 12), 85)	RAS	((12, 2), 122)	FCS	((40, 34), 14)	AMV**
((5, 1), 132)	NSL	((19, 14), 78)	VNT	((7, 3), 118)	CWS	((39, 35), 15)	AMV**
((6, 11), 107)	DGS	((19, 9), 108)	SPR	((5, 11), 126)	GUN	((38, 36), 11)	AMV**
((7, 12), 62)	CRN	((20, 3), 148)	BWS	((12, 7), 121)	OAV	((38, 37), 10)	AMV**
((12, 16), 74)	CDW	((22, 7), 67)	SWT	((12, 12), 58)	CRA	((38, 38), 9)	AMV**
((26, 13), 70)	ECS	((17, 9), 130)	FWS	((7, 11), 127)	TC1		
((26, 4), 59)	MGC	((13, 8), 130)	GWD	((3, 1), 129)	FIN		

* Outbound Transit Points

** Inbound Transit Points

This gives us the shortest path to complete all the trials and the time required to complete is 5,121 minutes. The resulting time provides us a lower bound on the time to conduct all trials provided all trials are successful in first attempt. The path generated can be plotted on the trial area grid. Since the trial area is small and number of trials is very large, a route plot for a smaller number of trials is shown in Figure 10.

Figure 10. Route Plot for 10 Trials.

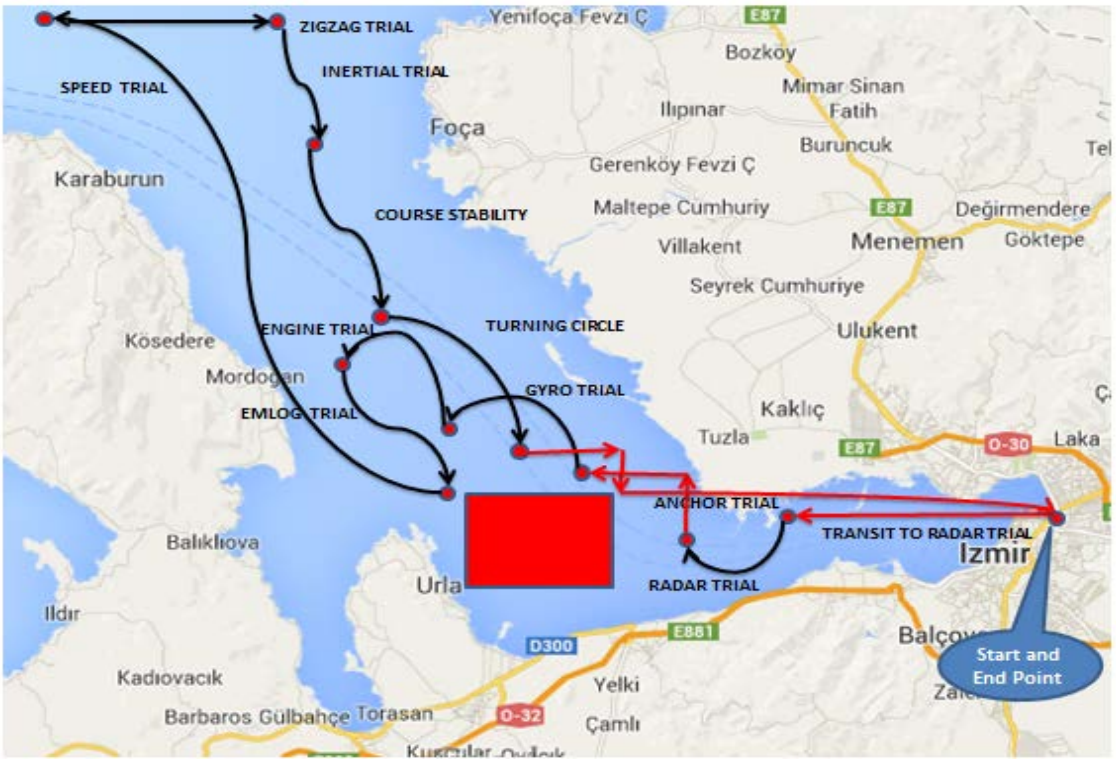
Background from Google Maps September 9, 2015, Retrieved from https://www.google.com/maps/place/Izmir,+%C4%B0zmir+Province,+Turkey/@38.4580005,26.968075,11z/data=!4m2!3m1!1s0x14bbd862a762cacd:0x628cbba1a59ce8fe

5. Out of Bound Areas

The program also has the provision to avoid certain areas at sea. If we identify some areas such as high fishing density areas or heavy shipping lanes that we have to avoid, they can be given as inputs and the resulting path will be generated while avoiding those areas. Small scale model runs with interdicted areas are shown in Figure 11. The

red box represents the out of bound area and the resulting path generated avoids that area while completing all ten trials.

Figure 11. Route Plot for 10 Trials with Interdiction Area.



Background from Google Maps September 9, 2015, Retrieved from https://www.google.com/maps/place/Izmir,+%C4%B0zmir+Province,+Turkey/@38.4580005,26.968075,11z/data=!4m2!3m1!1s0x14bbd862a762cacd:0x628cbba1a59ce8fe

B. SIMULATION AND DESIGN OF EXPERIMENTS

After generating the sequence and route for the trials, the process of conduct of trials was simulated in Python. The code used for running the simulations is at Appendix E. Experiments were designed varying five input factors. 51 design points were created in total which were reduced to 49 by removing the duplicate center points. 49 design points, generated by stacking and rotating the 17-run design from the NOLHdesigns_V6.xlsx spreadsheet, are shown in Table 5.

Table 5. Design Points for the Simulation, where the Factors are: *n* (Total Number of Acceptable Failures), *k* (Number of Retries per Trial), Wind State, Sea State, and Speed of the Ship.

Design Points 1-24					**Design Points 25-49**				
n	***k***	**wind**	**sea**	**speed**	***n***	***k***	**wind**	**sea**	**speed**
8	5	5	3	14	5	3	4	4	11
6	3	5	3	10	13	2	3	5	20
6	3	1	2	19	11	5	5	3	13
7	4	3	5	18	9	4	1	3	23
13	5	3	2	15	6	4	5	4	25
15	3	3	4	11	12	4	2	5	14
11	3	6	2	23	13	2	4	2	18
11	5	5	5	22	6	3	1	2	16
10	4	4	3	18	13	3	2	5	18
12	2	2	4	21	14	4	1	2	19
14	4	2	3	25	6	3	4	4	25
14	4	6	4	16	8	5	4	2	21
13	3	4	1	17	9	2	3	1	22
8	2	4	5	20	9	4	1	4	23
5	4	4	2	24	15	3	5	3	24
9	4	1	4	12	13	5	5	4	20
9	2	2	1	13	7	4	5	1	17
15	4	3	2	24	6	3	6	4	16
8	5	4	1	15	14	4	3	2	10
9	2	2	4	22	12	2	3	5	14
11	3	6	3	12	11	5	4	5	13
14	3	2	2	10	11	3	6	2	12
8	3	5	1	21	5	4	2	3	11
7	5	3	5	17	8	2	2	3	15
14	4	6	4	19					

The space filling provided by the Nearly Orthogonal Latin Hypercube design can be seen in Figure 12. The distribution of input factors along with summary statistics over the ranges specified in Table 1 and correlation matrix for input factors are shown in Figures 13 and 14 respectively. From Figure 13, we can see that although our design points are not uniformly distributed over the design space but they still cover the design space adequately. Also, absolute pairwise correlation between the factors is very low with the highest value of 0.1608 between *k* and *wind*. Minimum or no correlation is a desired property in the input factors.

Figure 12. Scatterplot Matrix of Input Factors.

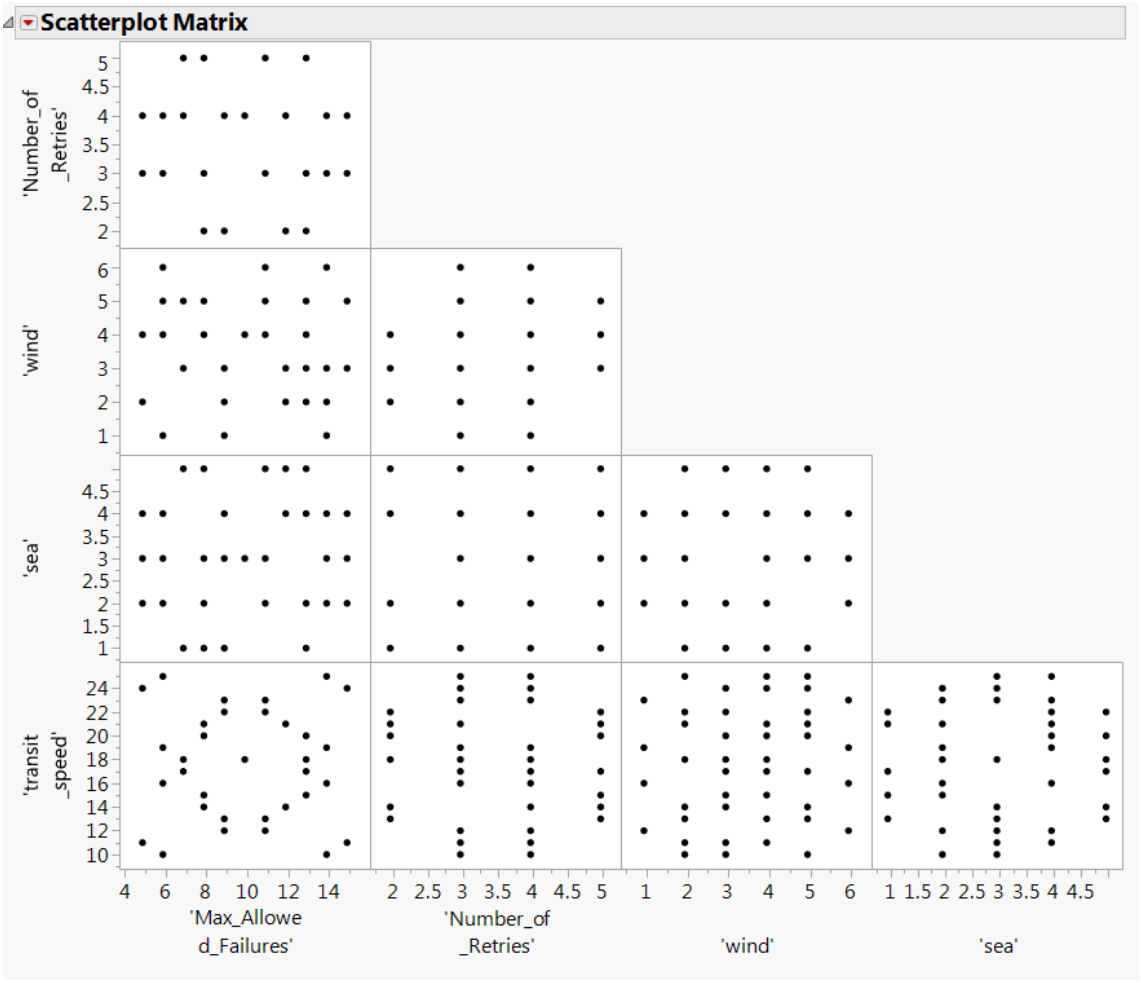


Figure 13. Distribution and Summary Statistics of Input Factors.

n		
100.0%	maximum	15
99.5%		15
97.5%		15
90.0%		14
75.0%	quartile	13
50.0%	median	10
25.0%	quartile	7.5
10.0%		6
2.5%		5
0.5%		5
0.0%	minimum	5

Summary Statistics	
Mean	10.061224
Std Dev	3.1318802
Std Err Mean	0.4474115
Upper 95% Mean	10.960806
Lower 95% Mean	9.1616435
N	49

k		
100.0%	maximum	5
99.5%		5
97.5%		5
90.0%		5
75.0%	quartile	4
50.0%	median	4
25.0%	quartile	3
10.0%		2
2.5%		2
0.5%		2
0.0%	minimum	2

Summary Statistics	
Mean	3.5102041
Std Dev	1.0025478
Std Err Mean	0.1432211
Upper 95% Mean	3.7981694
Lower 95% Mean	3.2222387
N	49

wind		
100.0%	maximum	6
99.5%		6
97.5%		6
90.0%		6
75.0%	quartile	5
50.0%	median	4
25.0%	quartile	2
10.0%		1
2.5%		1
0.5%		1
0.0%	minimum	1

Summary Statistics	
Mean	3.5102041
Std Dev	1.5827514
Std Err Mean	0.2261073
Upper 95% Mean	3.9648234
Lower 95% Mean	3.0555848
N	49

sea		
100.0%	maximum	5
99.5%		5
97.5%		5
90.0%		5
75.0%	quartile	4
50.0%	median	3
25.0%	quartile	2
10.0%		1
2.5%		1
0.5%		1
0.0%	minimum	1

Summary Statistics	
Mean	3.122449
Std Dev	1.3170778
Std Err Mean	0.188154
Upper 95% Mean	3.5007579
Lower 95% Mean	2.7441401
N	49

speed		
100.0%	maximum	25
99.5%		25
97.5%		25
90.0%		24
75.0%	quartile	21.5
50.0%	median	18
25.0%	quartile	13.5
10.0%		11
2.5%		10
0.5%		10
0.0%	minimum	10

Summary Statistics	
Mean	17.510204
Std Dev	4.6103256
Std Err Mean	0.6586179
Upper 95% Mean	18.834444
Lower 95% Mean	16.185964
N	49

Figure 14. Correlation Matrix for Input Factors

Correlations

	n	k	wind	sea	speed
n	1.0000	-0.0035	0.0188	0.0385	-0.0022
k	-0.0035	1.0000	0.1608	0.0464	-0.0214
wind	0.0188	0.1608	1.0000	-0.0206	-0.0079
sea	0.0385	0.0464	-0.0206	1.0000	0.0238
speed	-0.0022	-0.0214	-0.0079	0.0238	1.0000

C. MEASURES OF EFFECTIVENESS

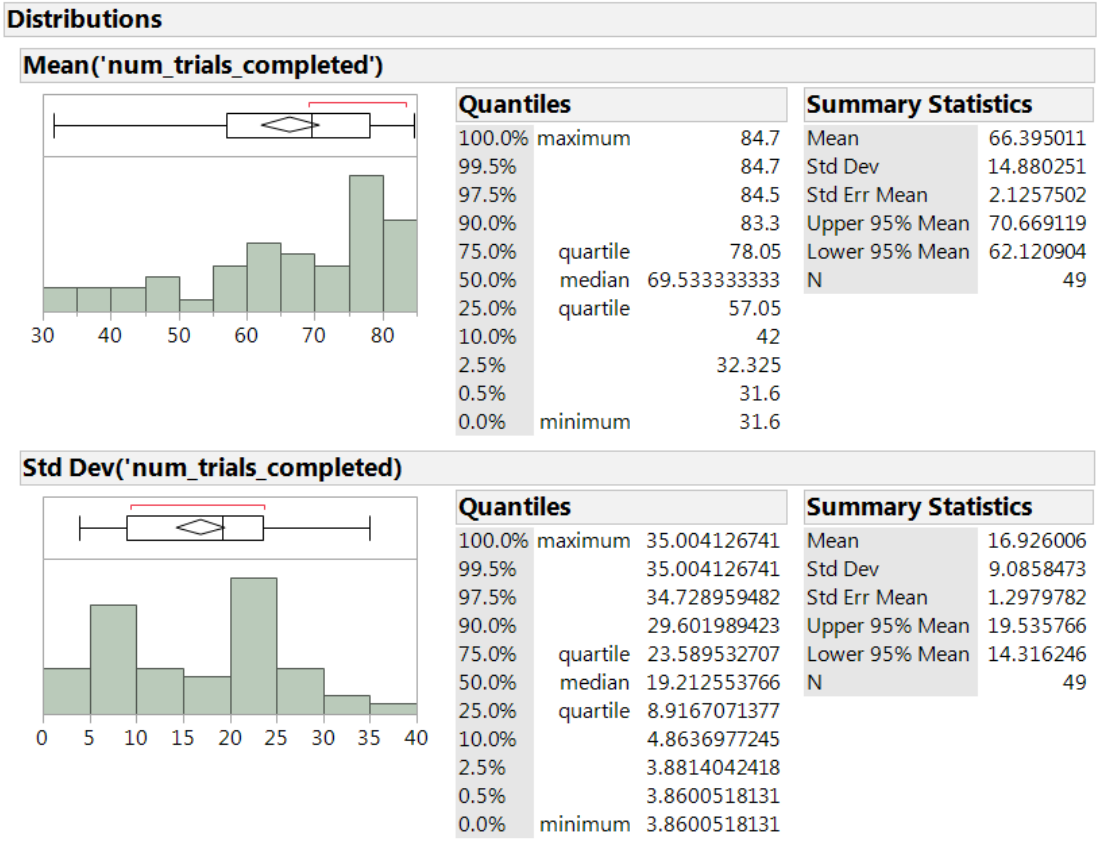
Two measures of effectiveness were chosen for the results, namely ‘total time’ and ‘number of trials completed.’ It was observed from the results that although we want a lower total time, however, simulation may yield lower times when we have more failures and thus very few trials completed. On the other hand, number of trials completed may be a better MOE as it is always desirable to complete maximum number

of trials. Therefore, 'number of trials completed' before returning to port was chosen as the primary MOE for the model.

D. SIMULATION RESULTS

The model uses design of experiments with 49 design points and 10 replications. The results are saved into a 490 row data set. This dataset is then condensed into a 49 row data set by calculating the mean and standard deviation for the number of trials completed for each design point. Figure 15 shows the histograms and summary statistics of the mean and standard deviation of number of trials completed collapsed over the input design factors space. We observe a mean of 66 trials completed with a standard deviation of 15.

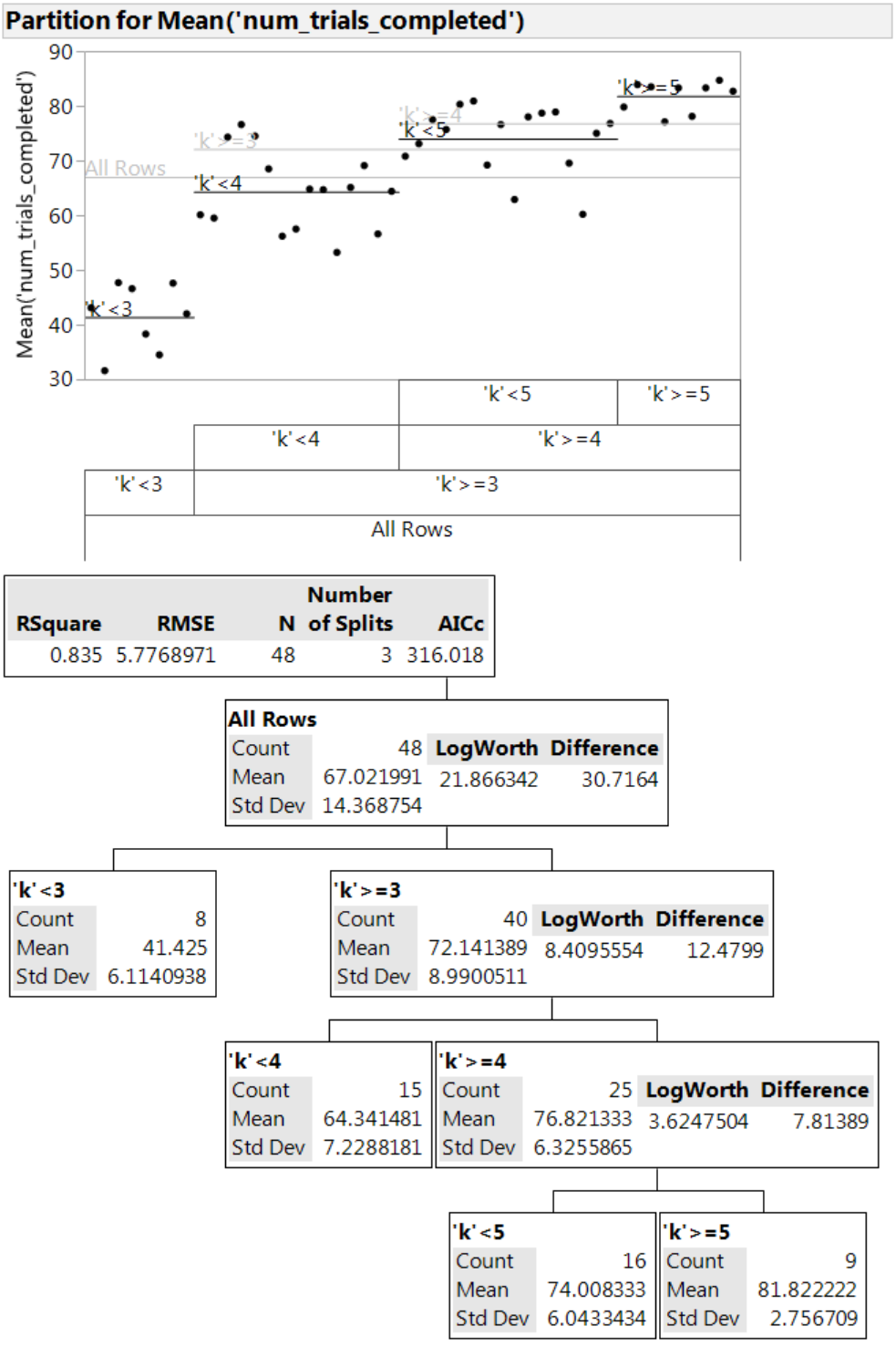
Figure 15. Histograms and Summary Statistics of Mean and Standard Deviation of Number of Trials Completed.



The results in Figure 15, together with a look at the raw data, show that although we are completing all the trials in some cases, there is no single design point that completes all trials in all ten replications.

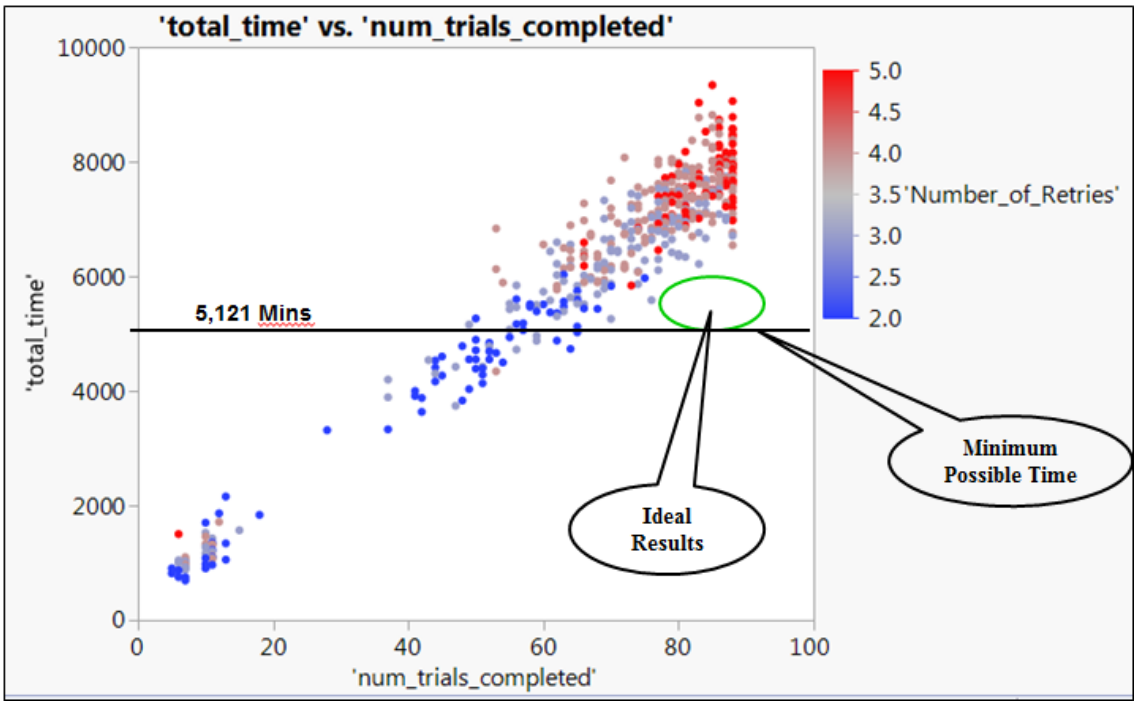
Next, we determine whether or not specific factors (or combinations of factors) are responsible for the differences in the MOE. One way of doing this is using a partition tree, which is a statistical method that recursively splits a large, heterogeneous data set into smaller, more homogeneous groups. When we fit a tree model for the mean number of trials completed as response variable, we find that number of retries per trial (k) is the most significant factor. The partition tree for number of trials completed is shown in Figure 16. With just three splits, it achieves an R^2 value of 0.835.

Figure 16. Partition tree for Mean Number of Trials Completed as Response Variable.



Based on this information, we observed that the number of retries per trial (*k*) is the most significant factor in the model. As the number of retries goes up, the number of trials completed also increases. None of the other input factors has a very high impact on the response. Figure 17 shows the scatter plot of total time against number of trials completed overlaid by number of retries (*k*). We observe that as the number of retries increases, we complete more trials. However, another important insight is the fact that none of the experiments result in ideal results where we complete all trials in the minimum possible time. This is owing to the failures experienced during the conduct of trials. Although there is an overall linear relationship between the two MOEs, we can a cluster of results in the bottom left corner which represents the cases where we did very poorly in terms of number of trials completed. Close inspection of these points shows us that in most of the cases these are either the cases where we have very low number of retries and acceptable failures, or very high sea state and wind force.

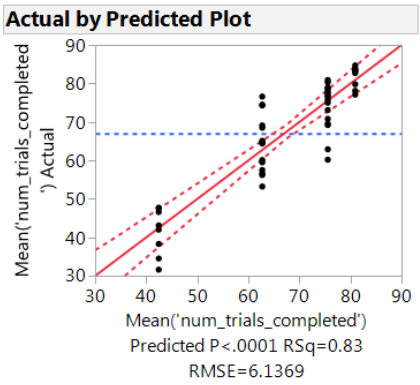
Figure 17. Total Time vs. Number of Trials Completed Overlaid by Number of Retries.



1. Regression Model

An alternative to a partition tree is a regression model. A tree may be better at capturing sudden jumps in the responses, but a regression model can be a simpler representation of the relationships of different parameters with the response. The regression model results for the mean number of trials completed as response variable are shown in Figure 18. We see that number of retries (*k*) comes out as the most significant factor.

Figure 18. Regression Summary for Mean Number of Trials Completed.



Summary of Fit

RSquare	0.82535
RSquare Adj	0.817588
Root Mean Square Error	6.136862
Mean of Response	67.02199
Observations (or Sum Wgts)	48

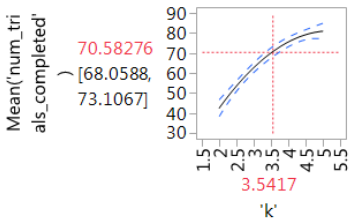
Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	2	8008.9234	4004.46	106.3289
Error	45	1694.7483	37.66	Prob > F
C. Total	47	9703.6717		<.0001*

Sorted Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
'k'	12.514261	0.906782	13.80	<.0001*
('k'-3.54167)*('k'-3.54167)	-3.72189	0.92662	-4.02	0.0002*

Prediction Profiler

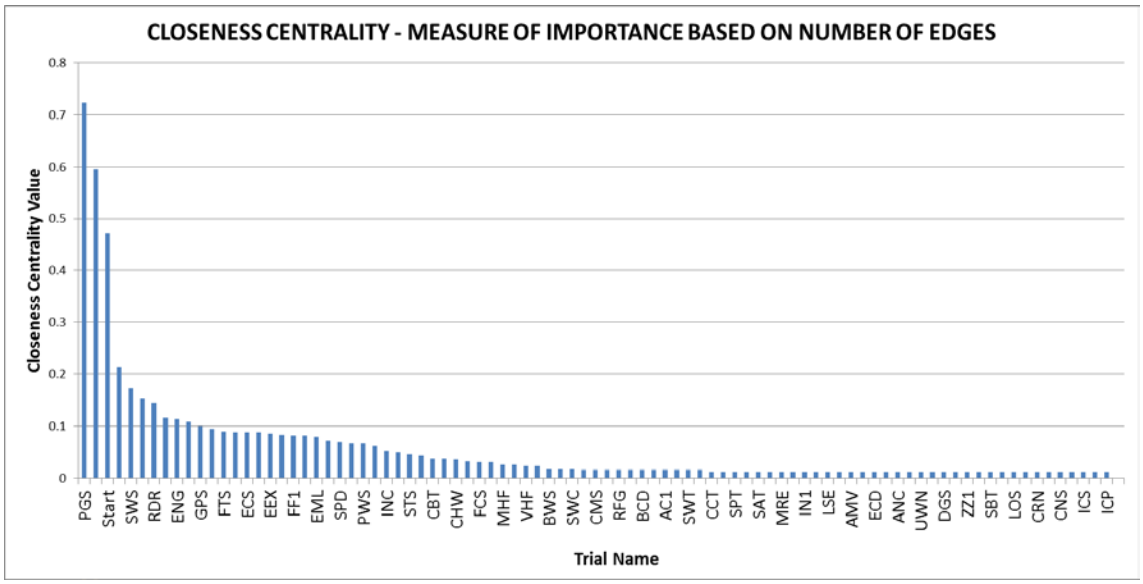


This model is statistically significant with an F-statistic of 106. The t-statistics and p-values verify that both the linear and quadratic terms are statistically significant (p-values < 0.0002). The R^2 of 0.82 is similar to that from the partition tree. It is quite high for a simple model, keeping in mind the fact that lack of fit is not a problem in this study as the objective is not to predict outcomes. The purpose is to identify significant variables and then apply that information to improve the process of conduct of trials.

2. Identification of Important Systems

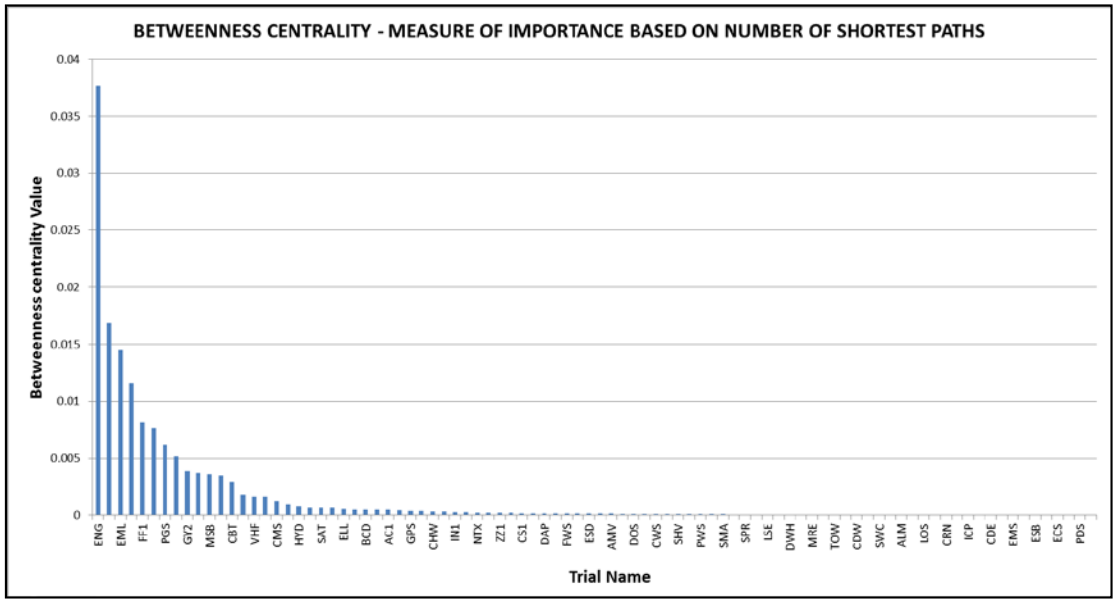
In order to identify the important systems in the network, we used centrality measures. We measured closeness centrality and between-ness centrality (Borgatti, 2005) for the nodes in the graph created for finding the sequence of trials. Closeness centrality measures the importance of a node based on the number of edges that node has. Figure 19 shows the plot of trials based on their closeness centrality values.

Figure 19. Closeness Centrality Plot.



Between-ness centrality measures the importance of a trial based on the number of shortest paths that trial occurs on. Figure 20 shows the plot of trials based on their between-ness centrality measures.

Figure 20. Between-ness Centrality Plot.



Based on the information from above two plots, we can identify the important systems in the network. Code used to calculate the closeness and between-ness centrality measures, along with the tables of closeness and between-ness centrality values for all systems, appear in Appendix F.

E. SIMULATION RE-RUNS

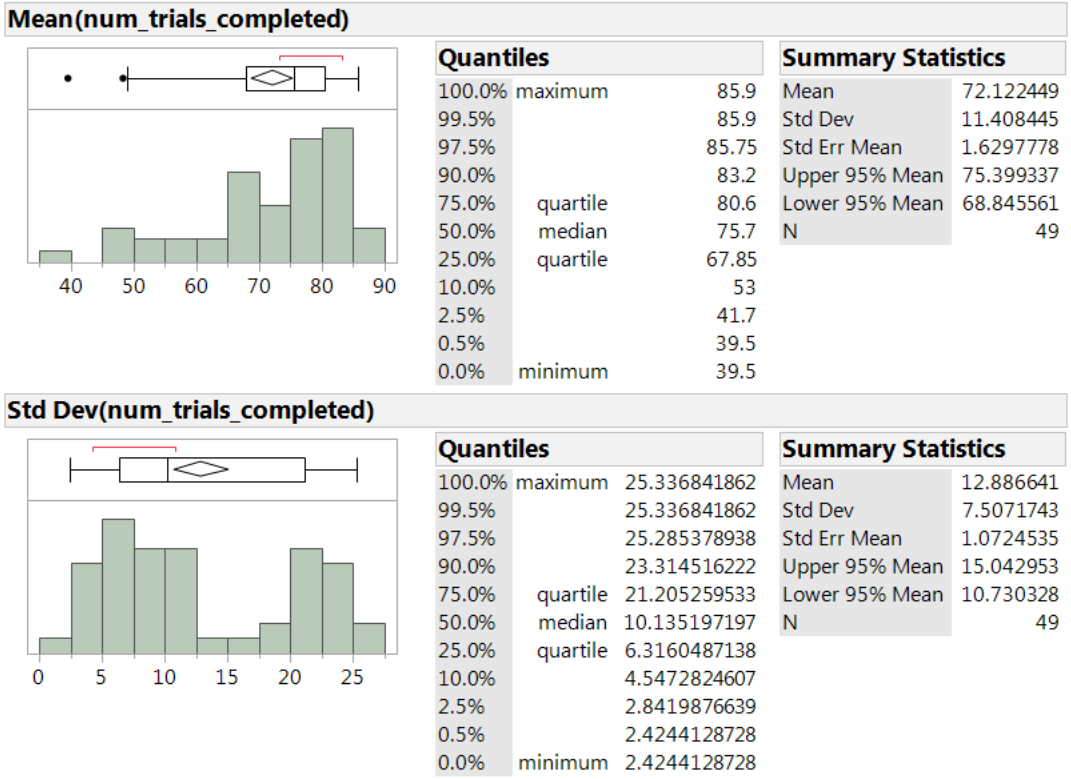
Based on the information about important systems and identifying the number of retries as the most significant factor, the simulation was re-run. It was decided that it may not be feasible to give all the systems extra retries in real life, as that may increase the total time for a single trip at sea to an unacceptably high level. Therefore, only the first ten important systems were given one extra retry. This extra retry to ten important systems represents availability of spares for those systems. There is limited room onboard ship for storage, therefore a limited amount of spares can be stored for sea trips. This is important for planners because ships undergoing builder trials do not have all the storage spaces operational. Therefore all the stores are brought on and off the ship before every sea trip.

The simulation was then re-run using same 49 design points as in initial simulation. Code used to re-run the simulation is in Appendix G.

1. Simulation Re-run Results

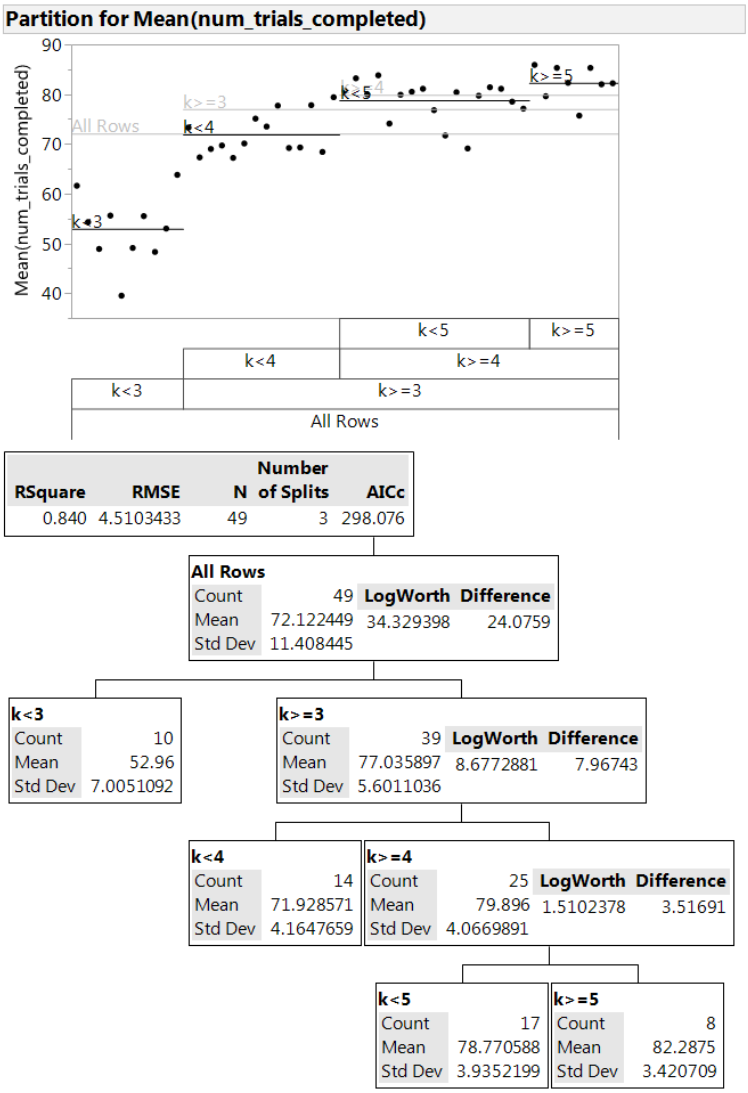
The model re-run uses design of experiments with 49 design points and ten replications. The results are saved into a 490 row data set. This dataset is then condensed into a 49 row data set by calculating the mean and standard deviation for the number of trials completed for each design point. Figure 23 shows the histogram and summary statistics of mean number of trials completed collapsed over the input design factors space. We observe a mean of 72 and standard deviation of 11 for mean number of trials completed.

Figure 21. Histograms and Summary Statistics of Mean and Standard Deviation of Number of Trials Completed.



Exploring the simulated data, we see that none of the design points completed all trials in all replications. However, our mean number of completed trials has increased and standard deviation of completed trials has decreased. The 25% quartile for the initial run had 58 completed trials, while in the simulation re-run the 25% quartile value increased to 67. The partition tree for the mean number of completed trials once again shows the number of retries (k) as the most significant factor. With three splits, this achieves an R^2 of 0.84. The partition tree is shown in Figure 24.

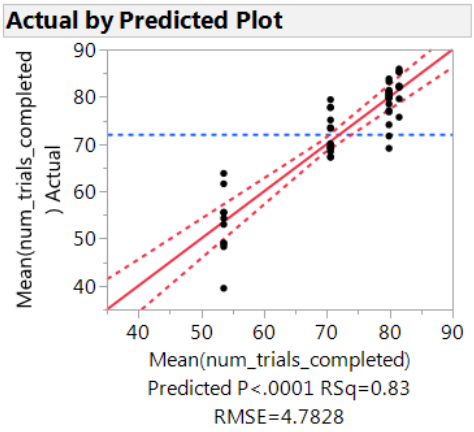
Figure 22. Partition Tree with Mean Number of Trials Completed as Response.



2. Regression Model for Simulation Re-run

The regression model results for the mean number of trial completed as response variable are shown in Figure 23. We see that number of retries (*k*) again comes out as the most significant factor.

Figure 23. Results of Regression Model.



Summary of Fit

RSquare	0.831567
RSquare Adj	0.824244
Root Mean Square Error	4.782798
Mean of Response	72.12245
Observations (or Sum Wgts)	49

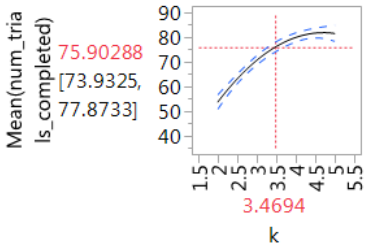
Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	2	5195.0680	2597.53	113.5526
Error	46	1052.2573	22.88	Prob > F
C. Total	48	6247.3253		<.0001*

Sorted Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
k	9.5386749	0.689475	13.83	<.0001*
(k-3.46939)*(k-3.46939)	-3.842734	0.712555	-5.39	<.0001*

Prediction Profiler

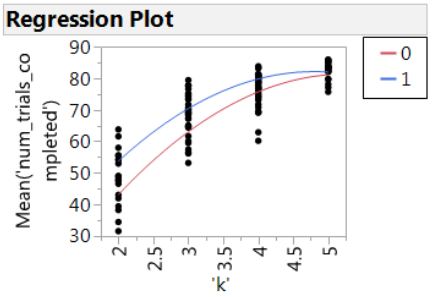


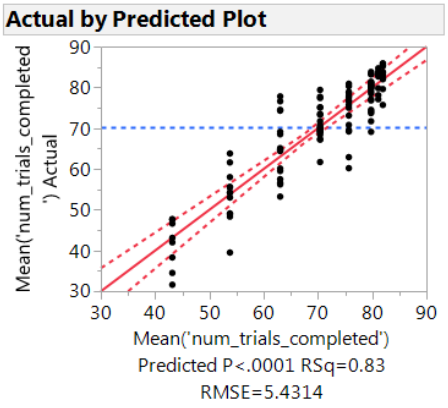
This model is statistically significant with an F-statistic of 113. The t-statistics and p-values verify that both the linear and quadratic terms are statistically significant (p-values < 0.0001). The R^2 of 0.83 is similar to that from the partition tree and is again quite high for a simple model, keeping in mind the fact that lack of fit is not a problem in this study as the objective is not to predict outcomes.

F. COMPARISON OF RESULTS

In order to compare the results from both the simulations, we concatenate the result tables into one table and add an indicator variable (I) such that $I = 0$ if results are from initial simulation and $I = 1$ otherwise. The initial simulation has k retries for each trial and the re-run has k+1 retries for 10 important trials and k retries for the others. After concatenating the tables, we fit a regression model to the complete data. Summary results from the regression model are shown in Figure 24.

Figure 24. Regression Model Results for Simulation Re-run.





Summary of Fit

RSquare	0.825229
RSquare Adj	0.818756
Root Mean Square Error	5.431389
Mean of Response	70.23766
Observations (or Sum Wgts)	113

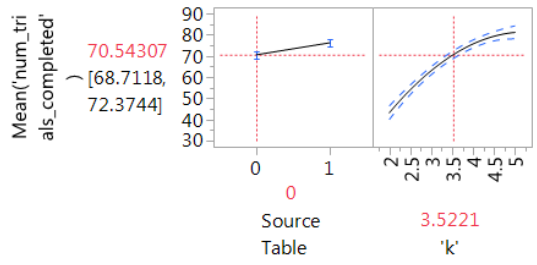
Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	4	15043.561	3760.89	127.4879
Error	108	3185.998	29.50	Prob > F
C. Total	112	18229.559		<.0001*

Parameter Estimates

Term	Estimate	Std Error	t Ratio	Prob>|t|
Intercept	35.127756	2.001408	17.55	<.0001*
Source Table[0]	-2.830233	0.515738	-5.49	<.0001*
'k'	10.858744	0.524516	20.70	<.0001*
Source Table[0]*('k'-3.52212)	1.6279149	0.524488	3.10	0.0024*
('k'-3.52212)*('k'-3.52212)	-3.613321	0.531783	-6.79	<.0001*

Prediction Profiler



From the regression plot in Figure 24, we can also see that mean number of completed trials is greater for simulation re-runs. Also the regression model is statistically significant with R^2 of 0.82 and an F-statistic of 127. Presence of the Indicator variable (named Source Table) shows that there is a statistically significant difference between the two data sets, and the interaction of the Source Table indicator value with *k* shows its impact interacts with the number of retries.

We also looked at both the data sets and analyzed the cases where all trials were completed. We observed that in initial runs, 8.3% of runs completed all trials; in simulation re-runs, 12% of the time all trials were completed. Relaxing the requirement for completion of all trials, we subset the data for cases where number of trials completed were greater than 70 (about 80% trial completion). Plots for number of trials completed vs. total time overlaid by number of retries for initial simulation and simulation re-run are shown in Figures 25 and 26.

Figure 25. Total Time vs. Number of Trials Completed, Overlaid by Number of Retries (*k*) when all Systems Given Same Number of Retries. Results of Subset Where Number of Trials Completed > 70. Simulation Runs with Lower Value of *k* not Seen Very Often.

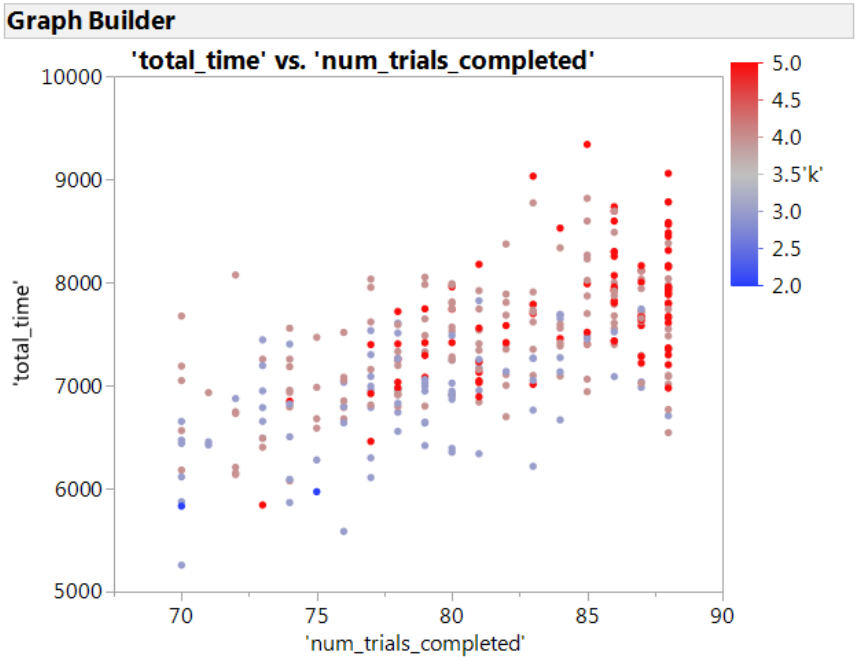
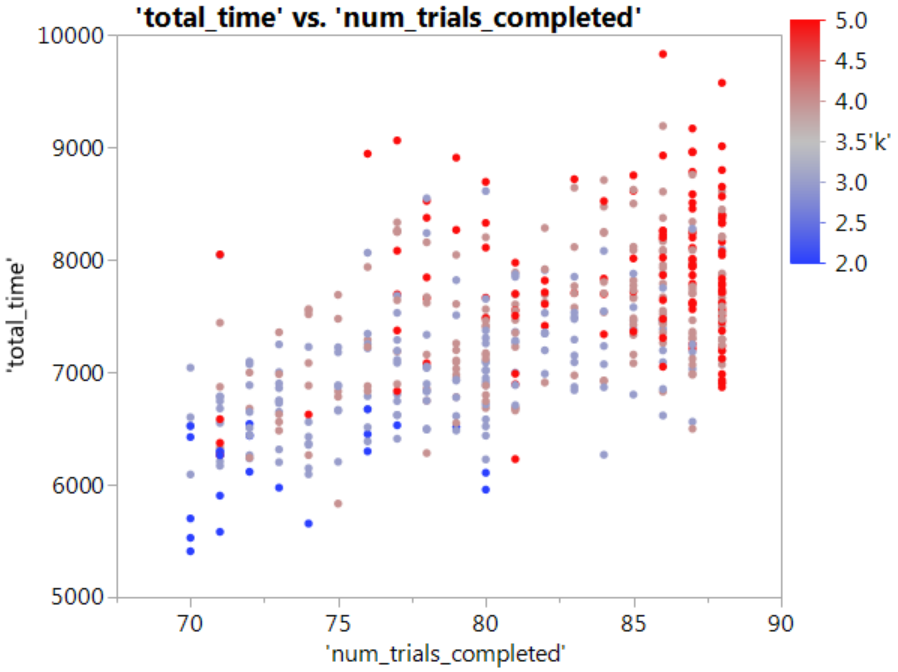


Figure 26. Total Time vs. Number of Trials Completed, Overlaid by Number of Retries (*k*), when One Extra Retry is Given to the Ten Most Important Systems. Results of Subset Where Number of Trials Completed > 70. Simulation Runs with Lower Value of *k* Completing More Trials.



G. DISCUSSION

This analysis demonstrates that identification of important systems improves the overall performance. Comparison of the graphs in Figures 25 and 26 shows that when we give one extra retry to important systems, we complete at least 70 trials more often even with lower numbers of overall retries (blue dots). Percentage-wise comparison of the data shows that there is an increase in number of cases where we complete at least 70 trials for one extra retry to important systems. The initial simulation has 61% of runs meet this threshold, while one extra retry for important systems increases it to 72%.

It is also observed that although the total number of acceptable failures is an important factor, it is not the primary factor affecting the number of trials completed. The number of retries given to each trial is the most significant factor during simulation runs. Identification of important systems can be translated into the spare supportability for those systems. During the conduct of builder trials when all the stores onboard ship are

not fully operational, spares need to brought on and off-board for each sea trip. If there is a failure at sea but we have spares for that system onboard, an attempt can be made to repair the system at sea. This, in essence, gives one extra retry for the conduct of the particular trial. It is evident from the simulation results that an extra retry improves the overall performance by leading to a greater likelihood of completing more trials. This is evident from the result comparison using the regression model shown in Figure 24.

Availability of this tool at sea for decision makers also provides them with useful information. In case of a failure of a system, the decision makers can re-run the sequence generator program, taking out all previously completed and previously failed trials. The new sequence generated will give them visibility about how many more trials can be completed by staying out at sea. This information will help the decision makers decide whether to continue with more trials or return to port. Moreover, the adaptive nature of the route generator will still provide the decision maker with an optimal route, provided he decides to stay out at sea and complete as many of the remaining trials as possible.

H. MODEL ENHANCEMENTS

This study shows that systematic planning improves the performance during conduct of builder trials for newly built ships. We have also gained useful insight on factors to be considered during planning and execution phase of the process. Although this study gives a concept of improvements in the planning process, it does not provide a user friendly tool at the moment. This problem can be solved by building a graphical user interface (GUI) for the model. By developing the GUI, the process will become very user friendly and may be easily used at sea when the situation arises. The GUI can be designed to take the input data from file provided by the user, read the area of trials from electronic chart system, and find the sequences and routes for the conduct. Future development may include plotting the geographical points representing the trial start location on the electronic chart system automatically. Also, the GUI may provide the user with the information about important systems in the network created by the user for the trial. This information will help the user better plan the spare availability onboard for important systems.

V. CONCLUSIONS AND RECOMMENDATIONS

Builder trials for newly built ships require a lot of money to be spent on the process. While initial planning can be done while in the port, uncertainties in outcomes make it difficult for the planners to capture all the possible outcomes during the initial planning phase. This necessitates the availability of an adaptive tool at sea that can help the decision makers in making the decision with regard to the conduct of further trials in case of any failure. Non-availability of such tool leaves the decision makers in a state of uncertainty, and as a result they often end up spending more time than required out at sea.

This thesis uses Python to develop a tool for the planning process of the builder trials for a newly built ship. The tool is used for generating the sequence for conduct of trials for all systems onboard along with the route for conduct of those trails. After generation of the sequence and route, the process of conduct of trials is simulated using a Python-based simulation model and design of experiments. The results from simulation are summarized and analyzed using linear regression and partition tree models. After identification of influential factors and important systems in the network, the simulation is re-run to evaluate the effectiveness of the process based on the finding from the first simulation runs. The primary MOE, total number of trials completed, improved significantly after model re-runs: from 66.2 to 72.1 of the potential 88 trials. This research is evidence that an efficient design can improve the overall performance during the conduct of builder trials. Moreover, the simulation process provides useful insight into the process, especially via identifying important systems and their impact on the overall results.

This study provides evidence that performance of a ship during builder trials can be improved by efficient planning of the process. Among the factors considered for the design of experiments in this study, we find number of retries given to a system (k) as the most influential factor for our MOE, number of trials completed. Based on these results following recommendations are made:

- The tool may be used in the initial planning phase for finding the sequence and route to conduct trials.

- In case of failure of a system at sea, the tool may be used to generate the alternate sequence for conduct of trials for remaining systems.
- The tool may be used for identification of important systems for spare supportability. This will in essence give the important systems one extra retry (k) in case failure.

The developed tool and the simulation model incorporate many assumptions. It may be beneficial to explore more scenarios while trying to remove the assumptions made.

Following are possible future works related to this research:

- Update the simulation program and analyze the results while catering for return to port and forcing the program to complete all trials. Presently, the simulation stops when a fixed number of acceptable failures are reached and the ship returns to port.
- Expand the scope by changing the distributions of the probability success to reflect appropriate historically collected data or values from requirements documents
- Improve ease-of-use for the navigator by changing position inputs from Cartesian coordinates to Latitude and Longitude for plotting.
- Analysis of multiple paths generated by simulation for insights regarding robustness of various path.

APPENDIX A. CONTENTS OF INPUT DATA FILE

Trial	ID	Time	Depth	Prerequisites	Speed	Mean	SD	Wind	Sea
engine	ENG	105	50	GY2; RDR; PGS; MSB; EGS; ESB; TLI; CO2; FF1; SWS; EMS; ECS; EEX	20	0.65	0.13	2	2
Fin Stabilizer	FIN	60	50	ENG	20	0.65	0.13	3	4
Compartment Noise	CNS	120	100	ENG; EEX; AC1; VNT; SWT; CHW; RFG; SWS; CAS; PGS	15	0.65	0.13	1	1
Domestic Appliances	DAP	60	20	GWD	5	0.65	0.13	5	4
Doors Windows Hatches	DWH	45	20		10	0.65	0.13	5	4
Crane	CRN	45	20	PGS; MSB	5	0.65	0.13	2	2
ICCP Equipment	ICP	30	20	PGS; MSB	10	0.65	0.13	4	3
Marine Growth Prevention System	MGP	30	20	PGS; MSB	10	0.65	0.13	4	3
Liquid Tank Level Indications	TLI	30	20		10	0.65	0.13	3	2
Alarm System	ALM	60	20	PGS; MSB	10	0.65	0.13	4	3
Engine Exhaust Flaps	EEX	45	50	EMS; ECS; PGS; MSB	20	0.65	0.13	4	2
FF System	FF1	90	20	CO2; SPR; SWS; FWS; PWS	10	0.65	0.13	4	3
Sprinkling System	SPR	60	50		10	0.65	0.13	4	3
CO2 Fire Extinguishing System	CO2	45	50		10	0.65	0.13	4	3
AC System	AC1	45	50	PGS; CHW; CDW; SWS; FWS; SWC	10	0.65	0.13	5	4
Ventilation System	VNT	30	50	PGS	10	0.65	0.13	2	4
Sewage	SWT	45	50	BWS; SWS; DOS	10	0.65	0.13	4	4

Treatment Plant									
Chilled Water System	CHW	45	50	PGS; SWS; CDW; MSB	10	0.65	0.13	4	4
Refrigeration System	RFG	60	50	PGS; CHW; SWS; FWS	10	0.65	0.13	4	4
Condensate Water System	CDW	60	50	PGS; MSB; SWS	10	0.65	0.13	4	4
Lube Oil System	LOS	45	50	PGS; MSB	10	0.65	0.13	4	2
Fuel Transfer System	FTS	30	20	PGS; MSB	10	0.65	0.13	4	2
Seawater System	SWS	60	30	PGS; MSB	10	0.65	0.13	4	2
Fresh Water System	FWS	45	50	PGS; MSB	10	0.65	0.13	4	2
Dirty Oil System	DOS	30	100	PGS; MSB	10	0.65	0.13	4	3
Bilge Water System	BWS	45	100	PGS; MSB; DOS; SWS	10	0.65	0.13	4	3
Pre-wetting System NBC	PWS	120	50	PGS; MSB; SWS; FWS	10	0.65	0.13	2	2
Compressed Air System	CAS	90	30	PGS; MSB	10	0.65	0.13	4	3
Power Generation System	PGS	120	20		10	0.65	0.13	4	3
Main Switchboard	MSB	60	20		10	0.65	0.13	4	3
Emergency Generator System	EGS	60	50	FTS; PGS; MSB	10	0.65	0.13	4	3
Emergency Switchboard	ESB	45	50	EGS; MSB; PGS	10	0.65	0.13	4	3
Power Distribution System	PDS	90	50	PGS; MSB; EGS; ESB	10	0.65	0.13	4	3

Battery Charging Discharging System	BCD	45	50	PGS; MSB; EGS; ESB; PDS	10	0.65	0.13	4	3
Electrical Lighting System	ELL	30	20	PGS; MSB; EGS; ESB; PDS	10	0.65	0.13	4	3
Sea Water Cooling System	SWC	30	50	PGS; MSB; SWS	10	0.65	0.13	4	3
Grey Water Collecting And Drainage System	GWD	30	100	PGS; MSB; SWS; FWS	10	0.65	0.13	4	3
Engine Monitoring System	EMS	60	50	PGS; MSB	20	0.65	0.13	4	3
Engine Control System	ECS	60	50	PGS; MSB	20	0.65	0.13	4	3
Hydraulic System	HYD	60	20	PGS; MSB	10	0.65	0.13	3	3
Degaussing System	DGS	45	100	PGS; MSB; SWS	15	0.65	0.13	2	3
Shaft Vibration Measurement	SHV	90	100	PGS; ENG	20	0.65	0.13	1	1
Aux Machinery Vibration Measurement	AMV	120	100	PGS; ENG; SWS	20	0.65	0.13	1	1
Underwater Noise Measurement	UWN	45	100	PGS; ENG; SPD	20	0.65	0.13	1	1
Overall Vibration Measurement	OAV	60	100	PGS; SPD; SWS; ENG; GUN	20	0.65	0.13	1	1
Gyro1	GY1	45	20	PGS	10	0.65	0.13	2	2
Gyro2	GY2	45	20	PGS; GY1	10	0.65	0.13	2	2
radar	RDR	45	20	PGS; GY2	10	0.65	0.13	2	2
EM Log	EML	45	30	GY2; ENG; RDR	15	0.65	0.13	1	1

zigzag	ZZ1	45	50	GY2; ENG; RDR; SPD	20	0.65	0.13	2	2
Turning Circle	TC1	30	50	RDR; GY2; ENG; SPD	20	0.65	0.13	2	2
Course Stability	CS1	45	50	ENG; GY2; SPD	20	0.65	0.13	2	2
Inertial	IN1	30	50	ENG; GY2; RDR; EML; SPD	15	0.65	0.13	2	2
Speed	SPD	120	50	ENG; GY2; RDR; EML; STS	20	0.65	0.13	2	2
Anchor	ANC	30	30	PGS; HYD	0	0.65	0.13	3	3
Mooring Equipment	MRE	30	30	PGS; HYD	10	0.65	0.13	2	2
Towing	TOW	90	50	PGS; HYD	5	0.65	0.13	3	3
Replenishment At Sea	RAS	120	50	PGS; HYD	15	0.65	0.13	3	3
Boat Launch And Recovery System	BLR	45	30	PGS; HYD	5	0.65	0.13	3	3
Sea Boat	SBT	60	30	BLR	5	0.65	0.13	3	3
Steering System	STS	90	50	GY2; RDR; SMA; GPS; MGC; ESD	15	0.65	0.13	2	2
Navigation & Signal Lights	NSL	30	50	PGS;MSB	5	0.65	0.13	2	2
Ship Manipulation Apparatus	SMA	45	50	PGS;MSB	5	0.65	0.13	4	3
Intercom System	ICS	60	30	PGS;MSB	5	0.65	0.13	4	3
Sound Powered Telephone	SPT	60	50	PGS;MSB;BCD	5	0.65	0.13	4	3
CCTV	CCT	30	30	PGS;MSB	5	0.65	0.13	4	3
Echo Sounder Shallow	ESS	45	20	PGS;MSB	10	0.65	0.13	2	2
Echo Sounder Deep	ESD	45	50	PGS;MSB	10	0.65	0.13	2	2
Integrated Navigation Console	INC	60	50	PGS; MSB; GY2; GY1; RDR; GPS; AIS; MGC; MTR; ESD	5	0.65	0.13	4	4
DGPS	GPS	45	20	PGS; MSB; GY2	10	0.65	0.13	4	4
AIS	AIS	30	30	PGS; MSB; GPS; GY2; ESD; EML	10	0.65	0.13	4	4
NAVTEX Rx	NTX	30	30	PGS; MSB; GPS; GY2; MTR	10	0.65	0.13	4	4

Metgraph	MTR	60	50	PGS; MSB;GPS	10	0.65	0.13	4	4
Magnetic Compass	MGC	120	50	PGS;MSB	5	0.65	0.13	3	2
VHF Comm	VHF	30	50	PGS; MSB; GY2; UHF	5	0.65	0.13	4	4
SATCOM	SAT	30	50	PGS;MSB;CMS	5	0.65	0.13	4	4
HF Emergency Radio	HFE	30	50		5	0.65	0.13	4	4
UHF Comm	UHF	30	30	PGS;MSB	5	0.65	0.13	4	4
MFHF Comm	MHF	30	50	PGS;MSB;VHF	5	0.65	0.13	4	4
Control And Distribution Eqpt	CDE	30	50	PGS;MSB;MHF	5	0.65	0.13	4	4
Communication System	CMS	60	50	PGS; MSB; CDE; MHF	5	0.65	0.13	4	4
WECDIS	ECD	60	20	GY1; GY2; EML; GPS; AIS; INC; ESD; ESS	10	0.65	0.13	4	4
Life Saving Equipment	LSE	60	50		5	0.65	0.13	3	2
Main Gun	GUN	120	50	FCS	10	0.65	0.13	3	4
CIWS	CWS	60	50	FCS	10	0.65	0.13	3	4
CRAA Guns	CRA	30	50	GY2;CBT	10	0.65	0.13	2	2
Combat System	CBT	180	50	INC	20	0.65	0.13	4	4
Fire Control System	FCS	60	50	INC;CBT	15	0.65	0.13	4	4

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. PYTHON FUNCTION TO GENERATE THE SEQUENCE OF TRIALS

```
import csv
import networkx as nx
import random

################################
#####Creating graph of trials###
################################

def create_graph(data_file):
    """This Function takes as input the data.csv file and creates a
    graph for finding the sequence to conduct trials"""
    csvr = csv.DictReader(open(data_file))
    g = nx.DiGraph()
    # add all the nodes
    for line in csvr:
        nid = line['id']
        nname = line['trial']
        requirements = line['reqs']
        time = int(line['time'])
        speed = line['speed']
        depth = line['depth']
        sea = line['sea']
        wind = line['wind']
        mean=float(line['mean'])
        sd=float(line['sd'])
        prob_success=float(round(random.normalvariate(mean,sd),3))
        g.add_node(nid, name= nname, requirements= requirements, time=
time, speed= speed, depth= depth, wind= wind, sea= sea, prob_success=
prob_success)

    # add all the edges
    for node,node_data in g.nodes(data=True):
        requirements = node_data['requirements']
        if requirements == '':
            continue
        for pred in requirements.split(';'):
            g.add_edge(pred,node,time=g.node[pred]['time'])
    # add the 'start' and 'end' nodes
    g.add_node('Start')
    g.add_node('End')
    for n in g.nodes():
        if n == 'Start' or n == 'End':
            continue
        if not g.predecessors(n):
            g.add_edge('Start',n,time=0)
        if not g.successors(n):
            g.add_edge(n,'End',time=-g.node[n]['time'])
    return g
#########################################################################
################################
```

```
###### Calling the function #####
#################################

g=create_graph("Data.csv")        #creating the trials graph
sort=nx.dag.topological_sort(g)  #generating a sequence for trials
#print sort
```

APPENDIX C. PYTHON CODE TO GENERATE MULTIPLE FEASIBLE SEQUENCES

```
from itertools import permutations, product, chain
import operator

########################################
### Generating the multiple sequences####
########################################

successors=nx.dfs_successors(g,'Start')
one_sequence=[x for x in sort if x in successors.keys()]
ofile = open('sequences_all.csv','w')
p=     {k:list(permutations(v))     for     k,     v     in
successors.iteritems()}
for seq in product(*map(p.get, one_sequence)):
    #seq='Start'+seq
    #print seq
    print >> ofile,list(chain.from_iterable(seq))
ofile.close() #writing the output to file
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. PYTHON CODE TO GENERATE THE PATH FOR TRIAL CONDUCT

```
import csv
import networkx as nx
import math
import random
from itertools import permutations, product, chain
from collections import deque
import operator
import numpy as np
import sys

############ Defining temporary variables ##################
total_time=0.0
distance=1.0 #distance in the grid (1 x 1 NM box)

#############################################
######## Grid Creator Function ##############
#############################################

def grid_creator(depth_file,area,area2,n_rows,n_cols):

    """Function to read the depths.csv file and create a grid.
    It also takes input for interdiction areas. It needs two lists of
    tuples for interdiction areas. e.g. [(29,32),(29,32)],[(1,3),(6,8)]
    represent two interdiction areas. If no interdiction is intended,
    give it coordinates outside ranges of depth file max rows and
    columns"""

    dt=depth_file
    target=open(dt,'r')

    Area=area#[(29,32),(29,32)]   # These areas for un-interdicted run
    Area2=area2#[(29,32),(29,32)]

    n_rows=n_rows
    n_cols=n_cols

    posn=[]
    d=[]
    for i in range(1,n_rows+1):
        line=target.readline()
        line=line.rstrip("\n")
        line=line.split(',')
        for j in range(1,n_cols+1):
            f=int(line[j-1])
            if f>0:
                if  i>=Area[0][0]  and  i<=Area[1][0]  and  j>=Area[0][1]
and j<=Area[1][1]:
                    continue
                elif    i>=Area2[0][0]    and    i<=Area2[1][0]    and
j>=Area2[0][1] and j<=Area2[1][1]:
```

```
                    continue
                else:
                    position=(i,j)
                    posn.append(position)
                    x=((i,j),f)
                    d.append(x)
    return d,posn
    target.close()

#####################################################################
#######MASTER NEIGHBORS DICT CREATION FUNCTION ##################
#####################################################################

def neighbours_creator(nodelist,output_dict_name):
    output_dict_name={}
    for nodes in nodelist:
        start=nodes
        #print start
        t1=(start[0]+1,start[1])  #North
        t2=(start[0]+1,start[1]+1) #North East
        t3=(start[0],start[1]+1) #East
        #t4=(start[0],start[1]) #self
        t5=(start[0]-1,start[1]+1) #South East
        t6=(start[0]-1,start[1]) #South
        t7=(start[0]-1,start[1]-1) #South West
        t8=(start[0],start[1]-1) #West
        t9=(start[0]+1,start[1]-1) #North West
        output_dict_name[start]=[]
        if t1 in nodelist:
            output_dict_name[start].append(t1)
        if t2 in nodelist:
            output_dict_name[start].append(t2)
        if t3 in nodelist:
            output_dict_name[start].append(t3)
        #if t4 in nodelist:
        #    output_dict_name[start].append(t4)
        if t5 in nodelist:
            output_dict_name[start].append(t5)
        if t6 in nodelist:
            output_dict_name[start].append(t6)
        if t7 in nodelist:
            output_dict_name[start].append(t7)
        if t8 in nodelist:
            output_dict_name[start].append(t8)
        if t9 in nodelist:
            output_dict_name[start].append(t9)
    return output_dict_name

#Example function call
#master_neighbors=neighbours_creator(posn,"master_neighbors")

#####################################################################
#############creating the graph for path checking ###################
#####################################################################
```

```
def path_checker_creator(depth_list,neighbors):
    path_checker = nx.DiGraph()
    path_checker.nodes=depth_list
    queue=[]
    index=0
    for key in neighbors.keys():
        #print "this is the key : %s"%(key,)
        #queue.append(key)
        #if key not in queue:
        queue.append(neighbors[key])
        #print     "These     are     the     neighbors     for     %s:
%s"%(key,neighbors50[key])
        for i in range(0,len(queue[index])):
            dest=queue[index][i]
            #print dest
            #type(dest)
            path_checker.add_edge(key,dest)
            #queue.pop()
            #queue=[]
        index+=1
    return path_checker

#########################################################################
#############stamping all nodes with layer information#################
#########################################################################

def node_stamper(d,layers):
    allnodes=[]
    for node in d:
    #print node
    #temp=[]
    #temp.append(d)
        for stamp in layers:
            if stamp is 'End':
                continue
        #print i
            else:
                tmp=[node,stamp]
                allnodes.append(tmp)
    return allnodes

#####################################################################
############ Layering on trial name ##############################
#####################################################################

#Creates  a  record  of  layers  and  adds  that  to  a  dictionary  for
referencing in main graph
def layer_record_creator(layers,allnodes):
    layer_record={}
    for layer in layers:
        layer_record[layer]=[]
        for node in allnodes:
            if node[1]==layer:
                layer_record[layer].append(node)
    return layer_record
```

```
#######################################################################
########## List of distance covered to complete a trial###########
#######################################################################

def trial_times_list(sort,g):
    cons=[]
    for i in sort:
        if i!='Start' and i!='End':
            m=int(g.node[i]['time'])*int(g.node[i]['speed'])/60.0
            cons.append(m)
    return cons

#######################################################################
########## List of depth required by each trial###################
#######################################################################

def depth_required_list(sort,g):
    depth_req=[]
    for i in sort:
        for node,node_data in g.nodes(data=True):
            if i==node and i!='Start' and i!='End':
                t=node_data['depth']
                depth_req.append(t)
    return depth_req

#######################################################################
########## Function to create the main routing graph #############
#######################################################################

def   main_graph_builder   (layer_record,   master_neighbors,   layers,
depth_req, sort, allnodes, cons, g, path_checker_all, path_checker_50,
path_checker_20):
    graph_1=nx.DiGraph()
    #print layer
    for layer in layers:
        ##print "This is the layer i am on: %s"%layer
        present_layer_nodes=layer_record[layer]
        #print present_layer_nodes
        for node in present_layer_nodes:
            source=tuple(node)
            #print node
            neighbors_complete=[node        for        neighbor        in
master_neighbors[node[0][0]]   for   node   in   present_layer_nodes   if
node[0][0]==neighbor]
            #print neighbors_complete
            for dest in neighbors_complete:
                dest=tuple(dest)
                #print "dest";print dest
                #print "writing edge from %s to %s"%(source,dest,)
                graph_1.add_edge(source,dest,time=(1.0*60/speedtr))
            neighbors_complete=[]
    for i in range(0,len(layers)):
        #print "This is trial number: %d"%i
        #print "This is trial: %s"%sort[i+1]
```

```
        if i<len(depth_req):
            depth_required=int(depth_req[i])
            #print "depth required updated to %d"%depth_required
        #if layers[i]!='End':

        present_layer=[node for node in allnodes if node[1]==sort[i] if
node[0][1]>=depth_required]
        #print "This is the present layer"
        #print present_layer
        next_layer=[node for node in allnodes if node[1]==sort[i+1] if
node[0][1]>=depth_required]
        #print "This is the next layer"
        #print next_layer

        for node1 in present_layer:
            temp1=node1[0][0]
            #print temp1
            x1=temp1[0]
            y1=temp1[1]
            source=tuple(node1)
            ##print "Source: %s"%source
            for node2 in next_layer:
                temp2=node2[0][0]
                x2=temp2[0]
                y2=temp2[1]
                dest=tuple(node2)
                ##print "Dest: %s"%dest
                if    (abs(x1-x2)+abs(y1-y2))<=cons[i]    and    (abs(x1-
x2)+abs(y1-y2))>=(cons[i]/2.0):
                    #print "distance is ok"

                    if layers[i]==layers[-1]:
                        #print "I m inside Start loop"
                        if nx.has_path(path_checker_all,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))

                    if layers[i]=='Start':
                        #print "I m inside Start loop"
                        if nx.has_path(path_checker_all,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
                    elif depth_required>=50:
                        #print "I am in depth 50 loop"
                        if nx.has_path(path_checker_50,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
```

```
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
                    elif depth_required>=20:
                        #print "I am in depth 20 loop"
                        if nx.has_path(path_checker_20,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
    return graph_1

######################################################################
########## Function to calculate results#############################
######################################################################

def result_calculator(graph_1,g,s,t):
    """This function takes the main graph, sequence graph, starting and
    ending locations as inputs and calculates shortest path and returns
    time to complete trials, path to be taken, betweenness and
    closeness centrality values"""

    #s=(((38, 38), 9), 'Start')
    #t=(((38, 38), 9), 'TC')
    result=nx.shortest_path(graph_1,s,t)
    total_time=0.0
    for i in range (0,len(result)):
        if result[i]!=result[-1]:
            if result[i][1]==result[i+1][1]:
                #print (result[i],result[i+1])
                total_time+=(1.0*60/speedtr)
            else:
                #print (result[i],result[i+1])
                total_time+=g.node[result[i+1][1]]['time']
    print "CLOSENESS CENTRALITY"
    print      sorted      (nx.closeness_centrality(g).items(),      key=
operator.itemgetter(1), reverse= True)
    print "BETWEENNESS CENTRALITY"
    print      sorted      (nx.betweenness_centrality(g).      items(),
key=operator.itemgetter(1), reverse=True)
    print "ROUTE FOR TRIALS"
    print result
    print "TIME TO COMPLETE TRIALS"
    print total_time

######################################################################
########### Execution of function calls ##############################
######################################################################

############################
##Command Line Arguments ##
############################
```

```
n=int(sys.argv[1])#10 #total number of failures allowed
k=int(sys.argv[2])#3 #number of retries per trial
wind=int(sys.argv[3])#3
sea=int(sys.argv[4])#2
speedtr=int(sys.argv[5])#20
edge_time=1.0*60/speedtr

#########################################################
######Reading depth file and creating grid##########
#########################################################

#change the input file name, areas 1 and 2 for interdicted runs and
n_rows and n_cols

d,posn=grid_creator("Depths.csv",[(29,32),(29,32)],[(29,32),(29,32)],42
,38)

######MASTER NEIGHBORS DICT CREATION ###############

master_neighbors=neighbours_creator(posn,"master_neighbors")

#####################################################################
################### Depth wise node lists #####################
#####################################################################

depth_all=[node[0] for node in d]
depth50=[node[0] for node in d if node[1]>=50] #d is the list of nodes
depth20=[node[0] for node in d if node[1]>=20]

##creating the dictionaries of neighbors depthwise
neighbors_all=neighbours_creator(depth_all,"neighbors_all")
neighbors50=neighbours_creator(depth50,"neighbors50")
neighbors20=neighbours_creator(depth20,"neighbors20")

#########################################################
#####creating the graph for path checking later ########
#########################################################

path_checker_all=path_checker_creator(depth_all,neighbors_all)
path_checker_20=path_checker_creator(depth20,neighbors20)
path_checker_50=path_checker_creator(depth50,neighbors50)

####################Creating graph of trials#################
g=create_graph("Data.csv")        #trials graph
sort=nx.dag.topological_sort(g)  #generating a sequence for trials
layers=sort[0:len(sort)-1]
sequence=deque(sort[1:len(sort)-1])

#####################################################################
########stamping all nodes with layer information##############
#####################################################################

allnodes=node_stamper(d,layers)
```

```
########################################
#### Layering on trial name###############
########################################

layer_record=layer_record_creator(layers,allnodes)

cons=trial_times_list(sort,g)

depth_req=depth_required_list(sort,g)

#### creating main graph ######

graph_1=
main_graph_builder(layer_record,master_neighbors,layers,depth_req,sort,
allnodes,cons,g,path_checker_all,path_checker_50,path_checker_20)

################################RESULTS###################################

s=(((38, 38), 9), layers[0]) #start Location
t=(((38, 38), 9), layers[-1]) #End Location
path,time=result_calculator_simulation(graph_1,g,s,t)
```

APPENDIX E. PYTHON CODE TO RUN SIMULATIONS

```
import csv
import networkx as nx
import math
import random
from itertools import permutations, product, chain
from collections import deque
import operator
import numpy as np
import sys

############ Defining temporary variables ##################

k_temp=0
n_temp=0
completed=[] #list of completed trials (keeps emptying)
failed=[] #list of failed trials (keeps emptying)
pending_trials=[] #list of pending trials
final_completed={} #dict with trials completed and number of attempts
final_failed=[] #list with failed trials
final_poor=[] #List with trials removed because of other trials failure
num_of_time_rescheduled=0
total_time=0.0
distance=1.0 #distance in the grid (1 x 1 NM box)

################################################
######## Grid Creator Function ##############
################################################

def grid_creator(depth_file,area,area2,n_rows,n_cols):

    """Function to read the depths.csv file and create a grid.
    It also takes input for interdiction areas. It needs two lists of
    tuples for interdiction areas. e.g. [(29,32),(29,32)],[(1,3),(6,8)]
    represent two interdiction areas. If no interdiction is intended,
    give it coordinates outside ranges of depth file max rows and
    columns"""

    dt=depth_file
    target=open(dt,'r')

    Area=area#[(29,32),(29,32)]   # These areas for un-interdicted run
    Area2=area2#[(29,32),(29,32)]

    n_rows=n_rows
    n_cols=n_cols

    posn=[]
    d=[]
    for i in range(1,n_rows+1):
        line=target.readline()
        line=line.rstrip("\n")
```

```
        line=line.split(',')
        for j in range(1,n_cols+1):
            f=int(line[j-1])
            if f>0:
                if  i>=Area[0][0]  and  i<=Area[1][0]  and  j>=Area[0][1]
and j<=Area[1][1]:
                    continue
                elif    i>=Area2[0][0]    and    i<=Area2[1][0]    and
j>=Area2[0][1] and j<=Area2[1][1]:
                    continue
                else:
                    position=(i,j)
                    posn.append(position)
                    x=((i,j),f)
                    d.append(x)
    return d,posn
    target.close()

######################################################################
#######MASTER NEIGHBORS DICT CREATION FUNCTION ##################
######################################################################

def neighbours_creator(nodelist,output_dict_name):
    output_dict_name={}
    for nodes in nodelist:
        start=nodes
        #print start
        t1=(start[0]+1,start[1])  #North
        t2=(start[0]+1,start[1]+1) #North East
        t3=(start[0],start[1]+1) #East
        #t4=(start[0],start[1]) #self
        t5=(start[0]-1,start[1]+1) #South East
        t6=(start[0]-1,start[1]) #South
        t7=(start[0]-1,start[1]-1) #South West
        t8=(start[0],start[1]-1) #West
        t9=(start[0]+1,start[1]-1) #North West
        output_dict_name[start]=[]
        if t1 in nodelist:
            output_dict_name[start].append(t1)
        if t2 in nodelist:
            output_dict_name[start].append(t2)
        if t3 in nodelist:
            output_dict_name[start].append(t3)
        #if t4 in nodelist:
        #    output_dict_name[start].append(t4)
        if t5 in nodelist:
            output_dict_name[start].append(t5)
        if t6 in nodelist:
            output_dict_name[start].append(t6)
        if t7 in nodelist:
            output_dict_name[start].append(t7)
        if t8 in nodelist:
            output_dict_name[start].append(t8)
        if t9 in nodelist:
            output_dict_name[start].append(t9)
```

```
    return output_dict_name

#Example function call
#master_neighbors=neighbours_creator(posn,"master_neighbors")

#########################################################################
#############creating the graph for path checking ######################
#########################################################################

def path_checker_creator(depth_list,neighbors):
    path_checker = nx.DiGraph()
    path_checker.nodes=depth_list
    queue=[]
    index=0
    for key in neighbors.keys():
        #print "this is the key : %s"%(key,)
        #queue.append(key)
        #if key not in queue:
        queue.append(neighbors[key])
        #print     "These     are     the     neighbors     for     %s:
%s"%(key,neighbors50[key])
        for i in range(0,len(queue[index])):
            dest=queue[index][i]
            #print dest
            #type(dest)
            path_checker.add_edge(key,dest)
            #queue.pop()
            #queue=[]
        index+=1
    return path_checker

################################
#####Creating graph of trials###
################################

def create_graph(data_file):
    """This Function takes as input the data.csv file and creates a
    graph for finding the sequence to conduct trials"""
    csvr = csv.DictReader(open(data_file))
    g = nx.DiGraph()
    # add all the nodes
    for line in csvr:
        nid = line['id']
        nname = line['trial']
        requirements = line['reqs']
        time = int(line['time'])
        speed = line['speed']
        depth = line['depth']
        sea = line['sea']
        wind = line['wind']
        mean=float(line['mean'])
        sd=float(line['sd'])
        prob_success=float(round(random.normalvariate(mean,sd),3))
```

```
        g.add_node(nid, name= nname, requirements= requirements, time=
time, speed= speed, depth= depth, wind= wind, sea= sea, prob_success=
prob_success)

    # add all the edges
    for node,node_data in g.nodes(data=True):
        requirements = node_data['requirements']
        if requirements == '':
            continue
        for pred in requirements.split(';'):
            g.add_edge(pred,node,time=g.node[pred]['time'])
    # add the 'start' and 'end' nodes
    g.add_node('Start')
    g.add_node('End')
    for n in g.nodes():
        if n == 'Start' or n == 'End':
            continue
        if not g.predecessors(n):
            g.add_edge('Start',n,time=0)
        if not g.successors(n):
            g.add_edge(n,'End',time=-g.node[n]['time'])
    return g

##########################################################################
#############stamping all nodes with layer information#################
##########################################################################

def node_stamper(d,layers):
    allnodes=[]
    for node in d:
    #print node
    #temp=[]
    #temp.append(d)
        for stamp in layers:
            if stamp is 'End':
                continue
        #print i
            else:
                tmp=[node,stamp]
                allnodes.append(tmp)
    return allnodes


#####################################################################
############ Layering on trial name ##############################
#####################################################################

#Creates a record of layers and adds that to a dictionary for
referencing in main graph
def layer_record_creator(layers,allnodes):
    layer_record={}
    for layer in layers:
        layer_record[layer]=[]
        for node in allnodes:
            if node[1]==layer:
```

```
                layer_record[layer].append(node)
    return layer_record

#######################################################################
########## List of distance covered to complete a trial###########
#######################################################################

def trial_times_list(sort,g):
    cons=[]
    for i in sort:
        if i!='Start' and i!='End':
            m=int(g.node[i]['time'])*int(g.node[i]['speed'])/60.0
            cons.append(m)
    return cons

#######################################################################
########## List of depth required by each trial###################
#######################################################################

def depth_required_list(sort,g):
    depth_req=[]
    for i in sort:
        for node,node_data in g.nodes(data=True):
            if i==node and i!='Start' and i!='End':
                t=node_data['depth']
                depth_req.append(t)
    return depth_req

#######################################################################
########## Function to create the main routing graph #############
#######################################################################

def   main_graph_builder   (layer_record,   master_neighbors,   layers,
depth_req, sort, allnodes, cons, g, path_checker_all, path_checker_50,
path_checker_20):
    graph_1=nx.DiGraph()
    #print layer
    for layer in layers:
        ##print "This is the layer i am on: %s"%layer
        present_layer_nodes=layer_record[layer]
        #print present_layer_nodes
        for node in present_layer_nodes:
            source=tuple(node)
            #print node
            neighbors_complete=[node        for        neighbor        in
master_neighbors[node[0][0]]   for   node   in   present_layer_nodes   if
node[0][0]==neighbor]
            #print neighbors_complete
            for dest in neighbors_complete:
                dest=tuple(dest)
                #print "dest";print dest
                #print "writing edge from %s to %s"%(source,dest,)
                graph_1.add_edge(source,dest,time=(1.0*60/speedtr))
            neighbors_complete=[]
    for i in range(0,len(layers)):
```

```
        #print "This is trial number: %d"%i
        #print "This is trial: %s"%sort[i+1]
        if i<len(depth_req):
            depth_required=int(depth_req[i])
            #print "depth required updated to %d"%depth_required
        #if layers[i]!='End':

        present_layer=[node for node in allnodes if node[1]==sort[i] if
node[0][1]>=depth_required]
        #print "This is the present layer"
        #print present_layer
        next_layer=[node for node in allnodes if node[1]==sort[i+1] if
node[0][1]>=depth_required]
        #print "This is the next layer"
        #print next_layer

        for node1 in present_layer:
            temp1=node1[0][0]
            #print temp1
            x1=temp1[0]
            y1=temp1[1]
            source=tuple(node1)
            ##print "Source: %s"%source
            for node2 in next_layer:
                temp2=node2[0][0]
                x2=temp2[0]
                y2=temp2[1]
                dest=tuple(node2)
                ##print "Dest: %s"%dest
                if   (abs(x1-x2)+abs(y1-y2))<=cons[i]   and   (abs(x1-
x2)+abs(y1-y2))>=(cons[i]/2.0):
                    #print "distance is ok"

                    if layers[i]==layers[-1]:
                        #print "I m inside Start loop"
                        if nx.has_path(path_checker_all,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))

                    if layers[i]=='Start':
                        #print "I m inside Start loop"
                        if nx.has_path(path_checker_all,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
                    elif depth_required>=50:
                        #print "I am in depth 50 loop"
                        if nx.has_path(path_checker_50,temp1,temp2):
```

```
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
                    elif depth_required>=20:
                        #print "I am in depth 20 loop"
                        if nx.has_path(path_checker_20,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
    return graph_1

######################################################################
########## Function to calculate results#############################
######################################################################

def result_calculator(graph_1,g,s,t):
    """This function takes the main graph, sequence graph, starting and
    ending locations as inputs and calculates shortest path and returns
    time to complete trials, path to be taken, betweenness and
    closeness centrality values"""

    #s=(((38, 38), 9), 'Start')
    #t=(((38, 38), 9), 'TC')
    result=nx.shortest_path(graph_1,s,t)
    total_time=0.0
    for i in range (0,len(result)):
        if result[i]!=result[-1]:
            if result[i][1]==result[i+1][1]:
                #print (result[i],result[i+1])
                total_time+=(1.0*60/speedtr)
            else:
                #print (result[i],result[i+1])
                total_time+=g.node[result[i+1][1]]['time']
    print "CLOSENESS CENTRALITY"
    print      sorted      (nx.closeness_centrality(g).items(),      key=
operator.itemgetter(1), reverse= True)
    print "BETWEENNESS CENTRALITY"
    print      sorted      (nx.betweenness_centrality(g).      items(),
key=operator.itemgetter(1), reverse=True)
    print "ROUTE FOR TRIALS"
    print result
    print "TIME TO COMPLETE TRIALS"
    print total_time

######################################################################
##### Function to calculate results for simulation ###################
######################################################################

def result_calculator_simulation(graph_1,g,s,t):
```

```
    """This function takes the main graph, sequence graph, starting and
ending locations
    as inputs and calculates shortest path and returns time to complete
trials, path to
    be taken, betweenness and closeness centrality values"""
    result=nx.shortest_path(graph_1,s,t)
    total_time=0.0
    for i in range (0,len(result)):
        if result[i]!=result[-1]:
            if result[i][1]==result[i+1][1]:
                total_time+=(1.0*60/speedtr)
            else:
                total_time+=g.node[result[i+1][1]]['time']
    return result,total_time

######################################################################
############## Function to simulate conduct of trial #############
######################################################################

def
conduct_trial(g,wind,sea,k,n,k_temp,n_temp,completed,failed,total_time,
trial,close_small,between_small,k_main):
    global final_completed, final_failed
    prob_success=g.node[trial]['prob_success']
    sea_req=g.node[trial]['sea']
    wind_req=g.node[trial]['wind']
    time=g.node[trial]['time']
    if wind<=wind_req and sea<=sea_req: #good wind and sea conditions
        #print "wind and sea ok"
        while k_temp<k:
            prob=random.random()
            if prob<=prob_success:
                #print "completed %s"%trial
                completed.append(trial)
                final_completed[trial]=k_temp
                k_temp=0
                total_time+=time
                break
            elif prob>=prob_success:
                #print "a try failed for %s"%trial
                k_temp+=1
                total_time+=time
            #break
        if k_temp>=k:
            n_temp+=1
            #print "failed %s %d times"%(trial,k_temp)
            failed.append(trial)
            final_failed.append(trial)
            k_temp=0
    else:  #Not favorable wind and sea conditions
        #print "wind sea not ok"
        prob_success=0.5*prob_success
        while k_temp<k:
            prob=random.random()
            if prob<=prob_success:
```

```
                #print "completed %s"%trial
                completed.append(trial)
                final_completed[trial]=k_temp
                k_temp=0
                total_time+=time
                break
            elif prob>=prob_success:
                #print "a try failed for %s"%trial
                k_temp+=1
                total_time+=time
            #break
        if k_temp>=k:
            n_temp+=1
            #print "failed %s %d times"%(trial,k_temp)
            failed.append(trial)
            final_failed.append(trial)
            k_temp=0
    return k_temp,n_temp,completed,failed,total_time

######################################################################
######## Function to update the graph after a failure ############
######################################################################

def graph_updater(g,flag=1):
#flag=1
    global failed,completed,final_poor
    while flag==1 and len(g.nodes())>2:
        for node,node_data in g.nodes(data=True):
            #print "node: %s"%node
            if node=='Start' or node=='End':
                continue

            else:
                flag=0
                if node in failed:
                    #print "removing: %s"%node
                    g.remove_node(node)  #remove  the  nodes  which  have
not been completed
                    flag=1
                elif node in completed:
                    #print "removing: %s"%node
                    g.remove_node(node)  #remove  the  nodes  which  have
not been completed
                    flag=1
        #for node_data in g.nodes(data=True):
            requirements=node_data['requirements'].split(';')
            for item in requirements: #iterate over the requirements of
all nodes
                #print "item: %s"%item
                if item in failed and node in g.nodes():
                    g.remove_node(node)
                    #print "inside removing: %s"%node
                    final_poor.append(node)
                    failed.append(node)
                    flag=1
```

```
    for n in g.nodes():
        if n == 'Start' or n == 'End':
            continue
        if not g.predecessors(n):
            g.add_edge('Start',n,time=0)
        if not g.successors(n):
            g.add_edge(n,'End',time=-g.node[n]['time'])
    failed=[]
    completed=[]
    return g

########################################################################
################## EXECUTION OF FUNCTIONS ##############################
########################################################################

n=int(sys.argv[1])#10 #total number of failures allowed
k=int(sys.argv[2])#3 #number of retries per trial
wind=int(sys.argv[3])#3
sea=int(sys.argv[4])#2
speedtr=int(sys.argv[5])#20
edge_time=1.0*60/speedtr
k_main=k

#######################################################
######Reading depth file and creating grid#########
#######################################################
#change the input file name, areas 1 and 2 for interdicted runs and
n_rows and n_cols

d,posn=grid_creator("Depths.csv",[(29,32),(29,32)],[(29,32),(29,32)],42
,38)

############MASTER NEIGHBORS DICT CREATION ##################

master_neighbors=neighbours_creator(posn,"master_neighbors")

########################################################################
################### Depth wise node lists ##############################
########################################################################

depth_all=[node[0] for node in d]
depth50=[node[0] for node in d if node[1]>=50] #d is the list of nodes
depth20=[node[0] for node in d if node[1]>=20]

##creating the dictionaries of neighbors depthwise
neighbors_all=neighbours_creator(depth_all,"neighbors_all")
neighbors50=neighbours_creator(depth50,"neighbors50")
neighbors20=neighbours_creator(depth20,"neighbors20")

########################################################################
######## creating the graph for path checking later###################
########################################################################

path_checker_all=path_checker_creator(depth_all,neighbors_all)
path_checker_20=path_checker_creator(depth20,neighbors20)
```

```
path_checker_50=path_checker_creator(depth50,neighbors50)

####################Creating graph of trials#####################

g=create_graph("Data.csv")        #trials graph
sort=nx.dag.topological_sort(g)  #generating a sequence for trials
layers=sort[0:len(sort)-1]
sequence=deque(sort[1:len(sort)-1])

#############################################################################
###################stamping all nodes with layer information) ##########
#############################################################################

allnodes=node_stamper(d,layers)

########################################
############ Layering on trial name#######
########################################

layer_record=layer_record_creator(layers,allnodes)

cons=trial_times_list(sort,g)

depth_req=depth_required_list(sort,g)

graph_1=main_graph_builder(layer_record,master_neighbors,layers,depth_r
eq,sort,allnodes,cons,g,path_checker_all,path_checker_50,path_checker_2
0)

################################RESULTS#####################################
######
s=(((38, 38), 9), layers[0])
t=(((38, 38), 9), layers[-1])
path,time=result_calculator_simulation(graph_1,g,s,t)

#transit_out_time= time to reach first point of trial
#transit_in_time= Time to reach back from last trial
final_route=[]
time_keeper=[]
list_of_trial_spots=deque([x  for  x  in  path  if  x[1]!=layers[0]  and
x[1]!=layers[-1]])
sea_bound_transit_route=[x for x in path if x[1]==layers[0]]
start_time=len(sea_bound_transit_route)*edge_time
time_keeper.append(start_time)
home_bound_transit_route=[x for x in path if x[1]==layers[-1]]
end_time=len(home_bound_transit_route)*edge_time
time_keeper.append(end_time)
transit_time=start_time+end_time
list_of_trial_spots.append(home_bound_transit_route[0])
for point in sea_bound_transit_route:
    final_route.append(point)

sequence=deque(sort[1:len(sort)-1])
k_temp=0
n_temp=0
```

```
completed=[] #list of completed trials (keeps emptying)
failed=[] #list of failed trials (keeps emptying)
pending_trials=[] #list of pending trials
final_completed={} #dictionary with trials completed and # of attempts
final_failed=[] #list with failed trials
final_poor=[]
num_of_times_rescheduled=0

s=path[0]
t=path[-1]
while n_temp<n and len(sequence)>0:
#    if len(sequence)>0:
    trial=sequence.popleft()
    #print "trial: %s"%trial
    s=list_of_trial_spots.popleft()

k_temp,n_temp,completed,failed,total_time=conduct_trial(g,wind,sea,k,n,
k_temp,n_temp,completed,failed,total_time,trial,close_small,between_sma
ll,k_main)
    #print "k_temp: %s"%k_temp
    #print "n_temp: %s"%n_temp
    #print "completed: %s"%completed
    #print "failed: %s"%failed
    #print "total_time: %d"%total_time
    if len(failed)<=0:
        final_route.append(s)
    if len(failed)>0:
        #print "Trial failed. Updating Graph"
        num_of_times_rescheduled+=1
        g=graph_updater(g,flag=1)
        sort=nx.dag.topological_sort(g)    #generating  a  sequence  for
trials
        #print "This is new sequence:%s"%sort
        layers=sort[0:len(sort)-1]
        sequence=deque(sort[1:len(sort)-1])
        #print "sequence: %s"%sequence
        ###### creating new network all over again
        allnodes=node_stamper(d,layers)
        layer_record=layer_record_creator(layers,allnodes)
        cons=trial_times_list(sort,g)
        depth_req=depth_required_list(sort,g)
        if len(depth_req)>0:

graph_1=main_graph_builder(layer_record,master_neighbors,layers,depth_r
eq,sort,allnodes,cons,g,path_checker_all,path_checker_50,path_checker_2
0)
            s=(s[0],layers[1])
            #print "This is s:%s"%(s,)
            t=(t[0],layers[-1])
            #print "This is t:%s"%(t,)
            path,time=result_calculator_simulation(graph_1,g,s,t)
            list_of_trial_spots=deque([x    for    x    in    path    if
x[1]!=layers[0] and x[1]!=layers[-1]])
            list_of_trial_spots.append(path[-len([x  for  x  in  path  if
x[1]==layers[-1]])])
```

```
            #sequence=deque(sort[1:len(sort)-1])
        else:
            print "All trials have been removed. Go back to port"

if n_temp<n:
    s=final_route[-1]
    t=(final_route[0][0],final_route[-1][1])
    home_bound_transit_route=nx.shortest_path(graph_1,s,t)
    for point in home_bound_transit_route:
        if point not in final_route:
            final_route.append(point)
    end_time=len(home_bound_transit_route)*edge_time
elif n_temp>=n and len(sequence)>0:
    s=(final_route[-1][0],sequence[0])
    t=(final_route[0][0],sequence[0])
    sequence.popleft()
    for trial in sequence:
        final_poor.append(trial)
    home_bound_transit_route=nx.shortest_path(graph_1,s,t)
    for point in home_bound_transit_route:
        if point not in final_route:
            final_route.append(point)
    end_time=len(home_bound_transit_route)*edge_time
transit_time=start_time+end_time
total_time+=transit_time
num_trials_failed=len(final_failed)
num_trials_completed=len(final_completed)
num_poor_trials=len(final_poor)#(len(g.nodes())-2)-
(len(final_completed.keys())+len(failed))  #trials  that  got  removed
because of others

#################################################################
############### Writing the output of simulation run ########
#################################################################

header="'n','k','wind','sea','speedtr','total_time','num_trials_failed'
,'num_trials_completed','num_poor_trials','final_failed_trials'"
print header
print
"%d,%d,%d,%d,%d,%d,%d,%d,%d,%s"%(n,k,wind,sea,speedtr,total_time,num_tr
ials_failed,num_trials_completed,num_poor_trials,final_failed)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. PYTHON CODE TO CALCULATE BETWEEN-NESS AND CLOSENESS CENTRALITY ALONG WITH RESULTS

```
close=sorted(nx.closeness_centrality(g).items(),
key=operator.itemgetter(1),reverse=True)

between=sorted(nx.betweenness_centrality(g).items(),
key=operator.itemgetter(1),reverse=True)
```

BETWEEN-NESS CENTRALITY RESULTS					
System	**Centrality Value**	**System**	**Centrality Value**	**System**	**Centrality Value**
‘ENG'	0.037643	'CHW'	0.000349	'LSE'	4.26E-05
'INC'	0.016855	'OAV'	0.000329	'HFE'	4.26E-05
'EML'	0.014492	'IN1'	0.000288	'DWH'	4.26E-05
'AIS'	0.011556	'CO2'	0.00026	'RAS'	3.77E-05
'FF1'	0.008172	'NTX'	0.000225	'MRE'	3.77E-05
'FCS'	0.007661	'TC1'	0.000224	'ANC'	3.77E-05
'PGS'	0.006139	'ZZ1'	0.000224	'TOW'	3.77E-05
'SPD'	0.005171	'TLI'	0.000196	'GY1'	2.89E-05
'GY2'	0.003919	'CS1'	0.000182	'CDW'	2.55E-05
'STS'	0.003703	'EEX'	0.000176	'CAS'	2.55E-05
'MSB'	0.003556	'DAP'	0.00017	'SWC'	2.55E-05
'EGS'	0.003447	'UWN'	0.000169	'CCT'	1.56E-05
'CBT'	0.002937	'FWS'	0.000153	'ALM'	1.56E-05
'MHF'	0.001788	'CRA'	0.000146	'MGP'	1.56E-05
'VHF'	0.00166	'ESD'	0.000144	'LOS'	1.56E-05
'CNS'	0.00165	'SPT'	0.000143	'NSL'	1.56E-05
'CMS'	0.001277	'AMV'	0.000138	'CRN'	1.56E-05
'ECD'	0.000912	'BWS'	0.000128	'ICS'	1.56E-05
'HYD'	0.000766	'DOS'	0.000128	'ICP'	1.56E-05
'GWD'	0.000681	'SBT'	0.000128	'VNT'	0
'SAT'	0.000654	'CWS'	0.000128	'CDE'	0
'GUN'	0.000638	'RFG'	0.000106	'End'	0
'ELL'	0.000526	'SHV'	0.000106	'EMS'	0
'BLR'	0.000511	'FIN'	9.98E-05	'FTS'	0
'BCD'	0.000511	'PWS'	8.51E-05	'ESB'	0
'SWT'	0.000511	'MGC'	7.87E-05	'Start'	0
'AC1'	0.000489	'SMA'	7.87E-05	'ECS'	0
'SWS'	0.000434	'ESS'	6.51E-05	'UHF'	0
'GPS'	0.000399	'SPR'	6.38E-05	'PDS'	0
'RDR'	0.000353	'DGS'	4.75E-05	'MTR'	0

CLOSENESS CENTRALITY RESULTS					
System	**Centrality Value**	**System**	**Centrality Value**	**System**	**Centrality Value**
PGS'	0.722736	'CBT'	0.036772	'MRE'	0.011236
'MSB'	0.594803	'SMA'	0.036404	'FIN'	0.011236
'Start'	0.470899	'CHW'	0.035955	'IN1'	0.011236
'GY2'	0.213067	'CDW'	0.031211	'CRA'	0.011236
'SWS'	0.173184	'FCS'	0.029963	'LSE'	0.011236
'GY1'	0.152949	'PDS'	0.029963	'CS1'	0.011236
'RDR'	0.144971	'MHF'	0.025682	'AMV'	0.011236
'EGS'	0.115366	'DOS'	0.025682	'SHV'	0.011236
'ENG'	0.112615	'VHF'	0.023408	'ECD'	0.011236
'ESB'	0.108507	'UHF'	0.022472	'TC1'	0.011236
'GPS'	0.098931	'BWS'	0.016854	'ANC'	0.011236
'FWS'	0.09472	'CDE'	0.016854	'MGP'	0.011236
'FTS'	0.08898	'SWC'	0.016854	'UWN'	0.011236
'EMS'	0.088714	'VNT'	0.014981	'HFE'	0.011236
'ECS'	0.088714	'CMS'	0.014981	'DGS'	0.011236
'CO2'	0.088714	'BLR'	0.014981	'ELL'	0.011236
'EEX'	0.084972	'RFG'	0.014981	'ZZ1'	0.011236
'ESD'	0.083261	'GWD'	0.014981	'NTX'	0.011236
'FF1'	0.082397	'BCD'	0.014981	'SBT'	0.011236
'TLI'	0.082397	'GUN'	0.014981	'TOW'	0.011236
'EML'	0.0799	'AC1'	0.014981	'LOS'	0.011236
'MGC'	0.07191	'CAS'	0.014981	'NSL'	0.011236
'SPD'	0.06882	'SWT'	0.014981	'CRN'	0.011236
'SPR'	0.066784	'ESS'	0.014981	'DAP'	0.011236
'PWS'	0.066784	'CCT'	0.011236	'CNS'	0.011236
'HYD'	0.061174	'OAV'	0.011236	'CWS'	0.011236
'INC'	0.051364	'SPT'	0.011236	'ICS'	0.011236
'MTR'	0.048852	'RAS'	0.011236	'DWH'	0.011236
'STS'	0.044944	'SAT'	0.011236	'ICP'	0.011236
'AIS'	0.043339	'ALM'	0.011236	'End'	0

APPENDIX G. PYTHON CODE FOR SIMULATION RE-RUNS

```
import csv
import networkx as nx
import math
import random
from itertools import permutations, product, chain
from collections import deque
import operator
import numpy as np
import sys

############ Defining temporary variables ##################

k_temp=0
n_temp=0
completed=[] #list of completed trials (keeps emptying)
failed=[] #list of failed trials (keeps emptying)
pending_trials=[] #list of pending trials
final_completed={} #dict with trials completed and number of attempts
final_failed=[] #list with failed trials
final_poor=[] #List with trials removed because of other trials failure
num_of_time_rescheduled=0
total_time=0.0
distance=1.0 #distance in the grid (1 x 1 NM box)

################################################
######## Grid Creator Function ##############
################################################

def grid_creator(depth_file,area,area2,n_rows,n_cols):

    """Function to read the depths.csv file and create a grid.
    It also takes input for interdiction areas. It needs two lists of
    tuples for interdiction areas. e.g. [(29,32),(29,32)],[(1,3),(6,8)]
    represent two interdiction areas. If no interdiction is intended,
    give it coordinates outside ranges of depth file max rows and
    columns"""

    dt=depth_file
    target=open(dt,'r')

    Area=area#[(29,32),(29,32)]   # These areas for un-interdicted run
    Area2=area2#[(29,32),(29,32)]

    n_rows=n_rows
    n_cols=n_cols

    posn=[]
    d=[]
    for i in range(1,n_rows+1):
        line=target.readline()
        line=line.rstrip("\n")
```

```
        line=line.split(',')
        for j in range(1,n_cols+1):
            f=int(line[j-1])
            if f>0:
                if  i>=Area[0][0]  and  i<=Area[1][0]  and  j>=Area[0][1]
and j<=Area[1][1]:
                    continue
                elif    i>=Area2[0][0]    and    i<=Area2[1][0]    and
j>=Area2[0][1] and j<=Area2[1][1]:
                    continue
                else:
                    position=(i,j)
                    posn.append(position)
                    x=((i,j),f)
                    d.append(x)
    return d,posn
    target.close()

######################################################################
#######MASTER NEIGHBORS DICT CREATION FUNCTION ##################
######################################################################

def neighbours_creator(nodelist,output_dict_name):
    output_dict_name={}
    for nodes in nodelist:
        start=nodes
        #print start
        t1=(start[0]+1,start[1])  #North
        t2=(start[0]+1,start[1]+1) #North East
        t3=(start[0],start[1]+1) #East
        #t4=(start[0],start[1]) #self
        t5=(start[0]-1,start[1]+1) #South East
        t6=(start[0]-1,start[1]) #South
        t7=(start[0]-1,start[1]-1) #South West
        t8=(start[0],start[1]-1) #West
        t9=(start[0]+1,start[1]-1) #North West
        output_dict_name[start]=[]
        if t1 in nodelist:
            output_dict_name[start].append(t1)
        if t2 in nodelist:
            output_dict_name[start].append(t2)
        if t3 in nodelist:
            output_dict_name[start].append(t3)
        #if t4 in nodelist:
        #    output_dict_name[start].append(t4)
        if t5 in nodelist:
            output_dict_name[start].append(t5)
        if t6 in nodelist:
            output_dict_name[start].append(t6)
        if t7 in nodelist:
            output_dict_name[start].append(t7)
        if t8 in nodelist:
            output_dict_name[start].append(t8)
        if t9 in nodelist:
            output_dict_name[start].append(t9)
```

```
    return output_dict_name

#Example function call
#master_neighbors=neighbours_creator(posn,"master_neighbors")

#########################################################################
#############creating the graph for path checking ######################
#########################################################################

def path_checker_creator(depth_list,neighbors):
    path_checker = nx.DiGraph()
    path_checker.nodes=depth_list
    queue=[]
    index=0
    for key in neighbors.keys():
        #print "this is the key : %s"%(key,)
        #queue.append(key)
        #if key not in queue:
        queue.append(neighbors[key])
        #print     "These     are     the     neighbors     for     %s:
%s"%(key,neighbors50[key])
        for i in range(0,len(queue[index])):
            dest=queue[index][i]
            #print dest
            #type(dest)
            path_checker.add_edge(key,dest)
            #queue.pop()
            #queue=[]
        index+=1
    return path_checker

################################
#####Creating graph of trials###
################################

def create_graph(data_file):
    """This Function takes as input the data.csv file and creates a
    graph for finding the sequence to conduct trials"""
    csvr = csv.DictReader(open(data_file))
    g = nx.DiGraph()
    # add all the nodes
    for line in csvr:
        nid = line['id']
        nname = line['trial']
        requirements = line['reqs']
        time = int(line['time'])
        speed = line['speed']
        depth = line['depth']
        sea = line['sea']
        wind = line['wind']
        mean=float(line['mean'])
        sd=float(line['sd'])
        prob_success=float(round(random.normalvariate(mean,sd),3))
```

```
        g.add_node(nid, name= nname, requirements= requirements, time=
time, speed= speed, depth= depth, wind= wind, sea= sea, prob_success=
prob_success)

    # add all the edges
    for node,node_data in g.nodes(data=True):
        requirements = node_data['requirements']
        if requirements == '':
            continue
        for pred in requirements.split(';'):
            g.add_edge(pred,node,time=g.node[pred]['time'])
    # add the 'start' and 'end' nodes
    g.add_node('Start')
    g.add_node('End')
    for n in g.nodes():
        if n == 'Start' or n == 'End':
            continue
        if not g.predecessors(n):
            g.add_edge('Start',n,time=0)
        if not g.successors(n):
            g.add_edge(n,'End',time=-g.node[n]['time'])
    return g

##########################################################################
#############stamping all nodes with layer information#################
##########################################################################

def node_stamper(d,layers):
    allnodes=[]
    for node in d:
    #print node
    #temp=[]
    #temp.append(d)
        for stamp in layers:
            if stamp is 'End':
                continue
        #print i
            else:
                tmp=[node,stamp]
                allnodes.append(tmp)
    return allnodes


#####################################################################
############ Layering on trial name ##############################
#####################################################################

#Creates a record of layers and adds that to a dictionary for
referencing in main graph
def layer_record_creator(layers,allnodes):
    layer_record={}
    for layer in layers:
        layer_record[layer]=[]
        for node in allnodes:
            if node[1]==layer:
```

```
                layer_record[layer].append(node)
    return layer_record

#######################################################################
########## List of distance covered to complete a trial###########
#######################################################################

def trial_times_list(sort,g):
    cons=[]
    for i in sort:
        if i!='Start' and i!='End':
            m=int(g.node[i]['time'])*int(g.node[i]['speed'])/60.0
            cons.append(m)
    return cons

#######################################################################
########## List of depth required by each trial###################
#######################################################################

def depth_required_list(sort,g):
    depth_req=[]
    for i in sort:
        for node,node_data in g.nodes(data=True):
            if i==node and i!='Start' and i!='End':
                t=node_data['depth']
                depth_req.append(t)
    return depth_req

#######################################################################
########## Function to create the main routing graph #############
#######################################################################

def   main_graph_builder   (layer_record,   master_neighbors,   layers,
depth_req, sort, allnodes, cons, g, path_checker_all, path_checker_50,
path_checker_20):
    graph_1=nx.DiGraph()
    #print layer
    for layer in layers:
        ##print "This is the layer i am on: %s"%layer
        present_layer_nodes=layer_record[layer]
        #print present_layer_nodes
        for node in present_layer_nodes:
            source=tuple(node)
            #print node
            neighbors_complete=[node        for        neighbor        in
master_neighbors[node[0][0]]   for   node   in   present_layer_nodes   if
node[0][0]==neighbor]
            #print neighbors_complete
            for dest in neighbors_complete:
                dest=tuple(dest)
                #print "dest";print dest
                #print "writing edge from %s to %s"%(source,dest,)
                graph_1.add_edge(source,dest,time=(1.0*60/speedtr))
            neighbors_complete=[]
    for i in range(0,len(layers)):
```

```
        #print "This is trial number: %d"%i
        #print "This is trial: %s"%sort[i+1]
        if i<len(depth_req):
            depth_required=int(depth_req[i])
            #print "depth required updated to %d"%depth_required
        #if layers[i]!='End':

        present_layer=[node for node in allnodes if node[1]==sort[i] if
node[0][1]>=depth_required]
        #print "This is the present layer"
        #print present_layer
        next_layer=[node for node in allnodes if node[1]==sort[i+1] if
node[0][1]>=depth_required]
        #print "This is the next layer"
        #print next_layer

        for node1 in present_layer:
            temp1=node1[0][0]
            #print temp1
            x1=temp1[0]
            y1=temp1[1]
            source=tuple(node1)
            ##print "Source: %s"%source
            for node2 in next_layer:
                temp2=node2[0][0]
                x2=temp2[0]
                y2=temp2[1]
                dest=tuple(node2)
                ##print "Dest: %s"%dest
                if   (abs(x1-x2)+abs(y1-y2))<=cons[i]   and   (abs(x1-
x2)+abs(y1-y2))>=(cons[i]/2.0):
                    #print "distance is ok"

                    if layers[i]==layers[-1]:
                        #print "I m inside Start loop"
                        if nx.has_path(path_checker_all,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))

                    if layers[i]=='Start':
                        #print "I m inside Start loop"
                        if nx.has_path(path_checker_all,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
                    elif depth_required>=50:
                        #print "I am in depth 50 loop"
                        if nx.has_path(path_checker_50,temp1,temp2):
```

```
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
                    elif depth_required>=20:
                        #print "I am in depth 20 loop"
                        if nx.has_path(path_checker_20,temp1,temp2):
                            #print  "There  is  a  path  between  %s  and
%s"%(temp1,temp2,)
                            #print "writing inter layer edge between %s
and %s"%(source,dest,)

graph_1.add_edge(source,dest,time=int(g.node[sort[i+1]]['time']))
    return graph_1

######################################################################
########## Function to calculate results#############################
######################################################################

def result_calculator(graph_1,g,s,t):
    """This function takes the main graph, sequence graph, starting and
    ending locations as inputs and calculates shortest path and returns
    time to complete trials, path to be taken, betweenness and
    closeness centrality values"""

    #s=(((38, 38), 9), 'Start')
    #t=(((38, 38), 9), 'TC')
    result=nx.shortest_path(graph_1,s,t)
    total_time=0.0
    for i in range (0,len(result)):
        if result[i]!=result[-1]:
            if result[i][1]==result[i+1][1]:
                #print (result[i],result[i+1])
                total_time+=(1.0*60/speedtr)
            else:
                #print (result[i],result[i+1])
                total_time+=g.node[result[i+1][1]]['time']
    print "CLOSENESS CENTRALITY"
    print      sorted      (nx.closeness_centrality(g).items(),      key=
operator.itemgetter(1), reverse= True)
    print "BETWEENNESS CENTRALITY"
    print      sorted      (nx.betweenness_centrality(g).      items(),
key=operator.itemgetter(1), reverse=True)
    print "ROUTE FOR TRIALS"
    print result
    print "TIME TO COMPLETE TRIALS"
    print total_time

######################################################################
##### Function to calculate results for simulation ###################
######################################################################

def result_calculator_simulation(graph_1,g,s,t):
```

```
    """This function takes the main graph, sequence graph, starting and
    ending locations as inputs and calculates shortest path and returns
    time to complete trials, path to be taken, betweenness and
    closeness centrality values"""

    result=nx.shortest_path(graph_1,s,t)
    total_time=0.0
    for i in range (0,len(result)):
        if result[i]!=result[-1]:
            if result[i][1]==result[i+1][1]:
                total_time+=(1.0*60/speedtr)
            else:
                total_time+=g.node[result[i+1][1]]['time']
    return result,total_time

######################################################################
############## Function to simulate conduct of trial #############
######################################################################

def
conduct_trial(g,wind,sea,k,n,k_temp,n_temp,completed,failed,total_time,
trial,close_small,between_small,k_main):
    global final_completed, final_failed
    #print "n_temp: %d"%n_temp
    #if len(sequence)>0:
    if trial in close_small:
        k+=1
        #print "updating k to value %d"%k
    elif trial in between_small:
        k+=1
        #print "updating k to value %d"%k
    else:
        k=k_main
    prob_success=g.node[trial]['prob_success']
    sea_req=g.node[trial]['sea']
    wind_req=g.node[trial]['wind']
    time=g.node[trial]['time']
    if wind<=wind_req and sea<=sea_req: #good wind and sea conditions
        #print "wind and sea ok"
        while k_temp<k:
            prob=random.random()
            if prob<=prob_success:
                #print "completed %s"%trial
                completed.append(trial)
                final_completed[trial]=k_temp
                k_temp=0
                total_time+=time
                break
            elif prob>=prob_success:
                #print "a try failed for %s"%trial
                k_temp+=1
                total_time+=time
            #break
        if k_temp>=k:
            n_temp+=1
```

```
            #print "failed %s %d times"%(trial,k_temp)
            failed.append(trial)
            final_failed.append(trial)
            k_temp=0
    else:  #Not favorable wind and sea conditions
        #print "wind sea not ok"
        prob_success=0.5*prob_success
        while k_temp<k:
            prob=random.random()
            if prob<=prob_success:
                #print "completed %s"%trial
                completed.append(trial)
                final_completed[trial]=k_temp
                k_temp=0
                total_time+=time
                break
            elif prob>=prob_success:
                #print "a try failed for %s"%trial
                k_temp+=1
                total_time+=time
            #break
        if k_temp>=k:
            n_temp+=1
            #print "failed %s %d times"%(trial,k_temp)
            failed.append(trial)
            final_failed.append(trial)
            k_temp=0
    k=k_main
    #print k
    return k_temp,n_temp,completed,failed,total_time

######################################################################
######## Function to update the graph after a failure ############
######################################################################

def graph_updater(g,flag=1):
#flag=1
    global failed,completed,final_poor
    while flag==1 and len(g.nodes())>2:
        for node,node_data in g.nodes(data=True):
            #print "node: %s"%node
            if node=='Start' or node=='End':
                continue

            else:
                flag=0
                if node in failed:
                    #print "removing: %s"%node
                    g.remove_node(node)  #remove  the  nodes  which  have
not been completed
                    flag=1
                elif node in completed:
                    #print "removing: %s"%node
                    g.remove_node(node)  #remove  the  nodes  which  have
not been completed
```

```
                    flag=1
        #for node_data in g.nodes(data=True):
            requirements=node_data['requirements'].split(';')
            for item in requirements: #iterate over the requirements of
all nodes
                #print "item: %s"%item
                if item in failed and node in g.nodes():
                    g.remove_node(node)
                    #print "inside removing: %s"%node
                    final_poor.append(node)
                    failed.append(node)
                    flag=1
    for n in g.nodes():
        if n == 'Start' or n == 'End':
            continue
        if not g.predecessors(n):
            g.add_edge('Start',n,time=0)
        if not g.successors(n):
            g.add_edge(n,'End',time=-g.node[n]['time'])
    failed=[]
    completed=[]
    return g

######################################################################
################### EXECUTION OF FUNCTIONS ###########################
######################################################################

n=int(sys.argv[1])#10 #total number of failures allowed
k=int(sys.argv[2])#3 #number of retries per trial
wind=int(sys.argv[3])#3
sea=int(sys.argv[4])#2
speedtr=int(sys.argv[5])#20
edge_time=1.0*60/speedtr
k_main=k

#######################################################
######Reading depth file and creating grid#########
#######################################################
#change the input file name, areas 1 and 2 for interdicted runs and
n_rows and n_cols

d,posn=grid_creator("Depths.csv",[(29,32),(29,32)],[(29,32),(29,32)],42
,38)

############MASTER NEIGHBORS DICT CREATION ##################

master_neighbors=neighbours_creator(posn,"master_neighbors")

######################################################################
################### Depth wise node lists ############################
######################################################################

depth_all=[node[0] for node in d]
depth50=[node[0] for node in d if node[1]>=50] #d is the list of nodes
depth20=[node[0] for node in d if node[1]>=20]
```

```
##creating the dictionaries of neighbors depthwise
neighbors_all=neighbours_creator(depth_all,"neighbors_all")
neighbors50=neighbours_creator(depth50,"neighbors50")
neighbors20=neighbours_creator(depth20,"neighbors20")

#########################################################################
######## creating the graph for path checking later###################
#########################################################################

path_checker_all=path_checker_creator(depth_all,neighbors_all)
path_checker_20=path_checker_creator(depth20,neighbors20)
path_checker_50=path_checker_creator(depth50,neighbors50)

####################Creating graph of trials#####################

g=create_graph("Data.csv")        #trials graph
sort=nx.dag.topological_sort(g)  #generating a sequence for trials
layers=sort[0:len(sort)-1]
sequence=deque(sort[1:len(sort)-1])

#########################################################################
###################stamping all nodes with layer information) ##########
#########################################################################

allnodes=node_stamper(d,layers)

##########################################
############ Layering on trial name#######
##########################################

layer_record=layer_record_creator(layers,allnodes)

cons=trial_times_list(sort,g)

depth_req=depth_required_list(sort,g)

graph_1=main_graph_builder(layer_record,master_neighbors,layers,depth_r
eq,sort,allnodes,cons,g,path_checker_all,path_checker_50,path_checker_2
0)

################################RESULTS#################################
######
s=(((38, 38), 9), layers[0])
t=(((38, 38), 9), layers[-1])
path,time=result_calculator_simulation(graph_1,g,s,t)

close=sorted(nx.closeness_centrality(g).items(),
key=operator.itemgetter(1),reverse=True)
close_small=[]
for item in close:
    if item[1]>=0.1:
        close_small.append(item[0])
#
```

```
between=sorted(nx.betweenness_centrality(g).items(),
key=operator.itemgetter(1),reverse=True)
between_small=[]
for item in between:
    if item[1]>=0.003:
        between_small.append(item[0])

#transit_out_time= time to reach first point of trial
#transit_in_time= Time to reach back from last trial
final_route=[]
time_keeper=[]
list_of_trial_spots=deque([x  for  x  in  path  if  x[1]!=layers[0]  and
x[1]!=layers[-1]])
sea_bound_transit_route=[x for x in path if x[1]==layers[0]]
start_time=len(sea_bound_transit_route)*edge_time
time_keeper.append(start_time)
home_bound_transit_route=[x for x in path if x[1]==layers[-1]]
end_time=len(home_bound_transit_route)*edge_time
time_keeper.append(end_time)
transit_time=start_time+end_time
list_of_trial_spots.append(home_bound_transit_route[0])
for point in sea_bound_transit_route:
    final_route.append(point)

sequence=deque(sort[1:len(sort)-1])
k_temp=0
n_temp=0
completed=[] #list of completed trials (keeps emptying)
failed=[] #list of failed trials (keeps emptying)
pending_trials=[] #list of pending trials
final_completed={} #dictionary with trials completed and # of attempts
final_failed=[] #list with failed trials
final_poor=[]
num_of_times_rescheduled=0

s=path[0]
t=path[-1]
while n_temp<n and len(sequence)>0:
#    if len(sequence)>0:
    trial=sequence.popleft()
    #print "trial: %s"%trial
    s=list_of_trial_spots.popleft()

k_temp,n_temp,completed,failed,total_time=conduct_trial(g,wind,sea,k,n,
k_temp,n_temp,completed,failed,total_time,trial,close_small,between_sma
ll,k_main)
    #print "k_temp: %s"%k_temp
    #print "n_temp: %s"%n_temp
    #print "completed: %s"%completed
    #print "failed: %s"%failed
    #print "total_time: %d"%total_time
    if len(failed)<=0:
        final_route.append(s)
    if len(failed)>0:
        #print "Trial failed. Updating Graph"
```

```
        num_of_times_rescheduled+=1
        g=graph_updater(g,flag=1)
        sort=nx.dag.topological_sort(g)    #generating  a  sequence  for
trials
        #print "This is new sequence:%s"%sort
        layers=sort[0:len(sort)-1]
        sequence=deque(sort[1:len(sort)-1])
        #print "sequence: %s"%sequence
        ###### creating new network all over again
        allnodes=node_stamper(d,layers)
        layer_record=layer_record_creator(layers,allnodes)
        cons=trial_times_list(sort,g)
        depth_req=depth_required_list(sort,g)
        if len(depth_req)>0:

graph_1=main_graph_builder(layer_record,master_neighbors,layers,depth_r
eq,sort,allnodes,cons,g,path_checker_all,path_checker_50,path_checker_2
0)
            s=(s[0],layers[1])
            #print "This is s:%s"%(s,)
            t=(t[0],layers[-1])
            #print "This is t:%s"%(t,)
            path,time=result_calculator_simulation(graph_1,g,s,t)
            list_of_trial_spots=deque([x    for    x    in    path    if
x[1]!=layers[0] and x[1]!=layers[-1]])
            list_of_trial_spots.append(path[-len([x  for  x  in  path  if
x[1]==layers[-1]])])
            #sequence=deque(sort[1:len(sort)-1])
        else:
            print "All trials have been removed. Go back to port"

if n_temp<n:
    s=final_route[-1]
    t=(final_route[0][0],final_route[-1][1])
    home_bound_transit_route=nx.shortest_path(graph_1,s,t)
    for point in home_bound_transit_route:
        if point not in final_route:
            final_route.append(point)
    end_time=len(home_bound_transit_route)*edge_time
elif n_temp>=n and len(sequence)>0:
    s=(final_route[-1][0],sequence[0])
    t=(final_route[0][0],sequence[0])
    sequence.popleft()
    for trial in sequence:
        final_poor.append(trial)
    home_bound_transit_route=nx.shortest_path(graph_1,s,t)
    for point in home_bound_transit_route:
        if point not in final_route:
            final_route.append(point)
    end_time=len(home_bound_transit_route)*edge_time
transit_time=start_time+end_time
total_time+=transit_time
num_trials_failed=len(final_failed)
num_trials_completed=len(final_completed)
```

```
num_poor_trials=len(final_poor)#(len(g.nodes())-2)-
(len(final_completed.keys())+len(failed))  #trials  that  got  removed
because of others

################################################################
############### Writing the output of simulation run ########
################################################################

header="'n','k','wind','sea','speedtr','total_time','num_trials_failed'
,'num_trials_completed','num_poor_trials','final_failed_trials'"
print header
print
"%d,%d,%d,%d,%d,%d,%d,%d,%d,%s"%(n,k,wind,sea,speedtr,total_time,num_tr
ials_failed,num_trials_completed,num_poor_trials,final_failed)
```

LIST OF REFERENCES

Borgatti, S. P. (2005). Centrality and network flow. Social networks, 27(1), 55–71. Retrieved from http://www.sciencedirect.com/science/article/pii/S0378873304000693.

Carter, J. M. (2005). *Shipbuilding integration* (Master's Thesis). Naval Postgraduate School, Monterey, CA. Retrieved from https://calhoun.nps.edu/bitstream/handle/10945/1841/05Dec_Carter.pdf?sequence=1&isAllowed=y

Cioppa, T. M., & Lucas, T. W. (2007). Efficient nearly orthogonal and space-filling Latin hypercubes. *Technometrics*, *49*(1), 45–55.

Colgary, K. A., & Willett, D. K. (2006). *Ship and Installation Program: Optimal Stationing of Naval Ships* (Master's Thesis). Naval Postgraduate School, Monterey, CA. Retrieved from

https://calhoun.nps.edu/bitstream/handle/10945/2768/06Jun_Colgary.pdf?sequence=1&isAllowed=y

Dimitrov, N. (2014, May). Network Flows and Graphs (instructional material). Presented at Naval Postgraduate School, Monterey, CA. Retrieved from http://neddimitrov.org/teaching/201402NFG.html

Germanischer Lloyd SE. (2012). *Rules for classification and construction* (VI-11-3). Hamburg, Germany: Germanischer Lloyd. Retrieved from http://www.gl-group.com/infoServices/rules/pdfs/gl_vi-11-3_e.pdf

Haakenstad, K. (2012). *Analysis and correction of sea trials* (Master's Thesis). Norweigan University of Science and Technology, Trondheim, Norway. Retrieved from http://brage.bibsys.no/xmlui/handle/11250/238219

Hart, C. (2000). Measurements during SWATH ship sea trials. *Instrumentation & Measurement Magazine*, IEEE, 3(3), 38–43. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=863910&tag=1

JMP, Version 12 (1989-2007). SAS Institute Inc., Cary, NC. Retrieved from https://www.nps.edu/Technology/SoftwareLib/Auth/index.htm

McLean, C., & Shao, G. (2001). Simulation in shipyards: simulation of shipbuilding operations. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer (Eds.), *Proceedings of the 2001 Winter Simulation Conference* (pp. 870–876). Piscataway, NJ: Institute of Electrical and Electronic Engineers. Retrieved from http://informs-sim.org/wsc01papers/114.pdf

The National Shipbuilding Research Program (NSRP) (1999, July), *Standard Ship Test and Inspection Plan, Procedures and Databases* (NSRP 0534 N6-95-1). U.S. Department of the Navy Carderock Division, Naval Surface Warfare Center. Retrieved from http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CB0QFjAAahUKEwj06Lqhmv3HAhUVNogKHVlNCw0&url=http%3A%2F%2Fhandle.dtic.mil%2F100.2%2FADA445496&usg=AFQjCNERUqqGdytt1YN5e_LxU8zrmw9I6w&sig2=6PD9q6Y6UEx_gY-h067zhw&bvm=bv.102829193,d.cGU

Sanchez, S. M., & Wan, H. (2012). Work smarter, not harder: A tutorial on designing and conducting simulation. In C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, & A. M. Urmacher (Eds.), *Proceedings of the 2012 Winter Simulation Conference* (pp. 1929–1943). Piscataway, NJ: Institute of Electrical and Electronic Engineers. Retrieved from http://informs-sim.org/wsc12papers/includes/files/inv260.pdf

Sanchez, S. M. (2011). NOLHDesigns_V6.xls [Spreadsheet file]. Retrieved from http://harvest.nps.edu

Vieira, H., Jr. (2012). NOB_Mixed_512DP_template_v1.xls [Spreadsheet file]. Retrieved from http://harvest.nps.edu

Vieira, H., Jr., Sanchez, S. M., Kienitz, K. H., & Belderrain, M. C. N. (2011). Improved efficient, nearly orthogonal, nearly balanced mixed designs. In S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, & M. Fu (Eds.), *Proceedings of the 2011 Winter Simulation Conference* (pp. 3605–3616). Piscataway, NJ: Institute of Electrical and Electronics Engineers. Retrieved from http://www.informs-sim.org/wsc11papers/320.pdf

Vieira, H., Jr., Sanchez, S. M., Kienitz, K. H., & Belderrain, M. C. N. (2013). Efficient nearly-orthogonal-and-balanced, mixed designs: An effective way to conduct trade-off analyses via simulation. *Journal of Simulation*, 7 (Special Issue on Input/Output Analysis), 264–275.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California